VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ VÝPOČTY VÍCENÁSOBNÝCH INTEGRÁLŮ

MULTIPLE INTEGRAL EFFECTIVE COMPUTATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. RADEK IŠA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VÁCLAV ŠÁTEK, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20100/2016/xisara00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Iša Radek, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Efektivní výpočty vícenásobných integrálů**
Multiple Integral Effective Computations
Kategorie: Modelování a simulace

Pokyny:

1. Seznamte se s matematickým postupem využívajícím metodu Taylorovy řady pro řešení diferenciálních rovnic.
2. Analyzujte paralelní vlastnosti metody Taylorovy řady a specifikujte matematické výpočty, které mohou být prováděny paralelně v samostatných procesorech.
3. Navrhněte a implementujte program pro výpočet vícenásobných integrálů, pro různé diferenční výrazy prostorové proměnné.
4. Navrhněte architekturu specializovaného výpočetního systému pro efektivní výpočty vícenásobných integrálů - včetně řízení výpočtu, s využitím různých diferenčních výrazů prostorové proměnné. Navrženou architekturu implementujte v provedení FPGA.
5. Činnost navrženého výpočetního systému ověřte na vhodných příkladech a určete možné zrychlení ve srovnání s klasickými metodami.

Literatura:

- Kalas, J., Kuben, J.: Integrální počet funkcí více proměnných, Brno 2006, Masarykova univerzita, Přírodovědecká fakulta
- Hirayama, H.: Fast Numerical Integration Using Taylor Series, in Integral Methods in Science and Engineering, Springer 2008
- další dle pokynů vedoucího

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Šátek Václav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Předkládaná práce se zabývá návrhem systému pro výpočet vícenásobných integrálů pro různé diferenční výrazy prostorové proměnné. V dnešní době je výpočet integrálů jedním z důležitých problémů inženýrství. Čtenář je nejdříve seznámen s různými metodami výpočtu integrálu. Následně je seznámen s numerickou integrací a využitím Taylorova rozvoje v numerické integraci. Praktickým cílem této práce je návrh softwarového a hardwarového systému pro výpočet vícenásobných integrálů.

Abstract

This thesis deals with the design system for multiple integrals for diferential expression with space variables. Today, integration is one of engineering problems. Reader is acquainted with different method of integration, then with numerican integration and Taylor series. The practical aim of this work is to design software and hardware system of numerican integration multiple integrals.

Klíčová slova

numerická integrace, Taylorova řada, diferenciální rovnice, integrátor, Boothův algoritmus,

Keywords

numeric integration, Taylor series, differential equeation, integrator, Booth algorithm

Citace

IŠA, Radek. *Efektivní výpočty vícenásobných integrálů.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Václav Šátek, Ph.D.

Efektivní výpočty vícenásobných integrálů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Václava Šátka Ph.D. Další informace mi poskytli doc. Ing. Jiří Kunovský, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Iša
23. května 2017

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce ing. Václavu Šátkovi Ph.D. a panu doc. Ing. Jiřímu Kunovskému, Csc., kteří mi poskytli důležité rady, náměty a pomohli mi cennými připomínkami. Mé poděkování patří také kolegům, rodině a kamarádům, kteří mě po dobu studia jakoliv podporovali a bez nichž by tato práce nemohla vzniknout.

Obsah

Kapitola 1

Úvod

Integrální a diferenciální počet je využíván ve velkém množství odvětví jako je matematika, fyzika, chemie, statika a mnoho dalších aplikací. Integrální a diferenciální rovnice mohou představovat nepřeberné množství abstrakcí z reálného světa. V současné době existují sice velmi rozsáhle znalosti analytického řešení integrálů, avšak dnes je potřeba počítat velmi složité a komplexní integrály, pro které je analytické řešení příliš pracné a složité. Dokonce v některých případech nejsou ani v současné době analyticky řešitelné. Tady vyvstává potřeba tyto příliš složité integrály vyčíslit numerickými metodami.

Velmi vážným problémem numerického řešení je vznik chyb způsobený nepřesností aproximace. Při velmi komplexních a rozměrných problémech může být chyba natolik veliká, že výsledek je prakticky nepoužitelný. Mohou také vznikat chyby závislé na velikosti datového typu. Jsou to takzvané chyby vzniklé ořezáním nebo zaokrouhlením čísel. Se zvětšujícím se rozměrem integrálu se zvětšuje i velikost výsledné chyby numerického výpočtu integrálu. Pro příliš velké integrály, pro které klasické metody dávají velice nepřesné výsledky, se používá numerická metoda Monte Carlo, která se snaží nepřesnost numerické integrace zvrátit určitou mírou náhodnosti ve výběru vzorků integrované funkce.

Častým využitím integrálního a diferenciálního počtu je simulace nejrůznějších problémů. Simulace se využívá k předurčení pravděpodobného chování fyzikálních, chemických, a jiných jevů. Využití bývá při určení pružnosti a pevnosti materiálů nebo k předpovězení chování elektrického obvodu. Proto je simulace důležitou součástí vývoje moderních technologií. Její pozitivní vliv na snižování ceny vyvíjeného produktu je nezanedbatelný. Využitím simulace je možné částečně otestovat výsledný produkt, aniž by bylo nutné daný výrobek vyrobit, což ušetří nemalé množství nákladů. Přesnost simulací je limitována metodami v simulaci použitými. Jak jsem již uvedl dříve, některé simulace využívají integrální a diferenciální počet, proto jehož zpřesněním či zrychlením dojde k zpřesnění či zrychlení těchto simulací. Jakékoliv vylepšení simulace má pozitivní vliv na vývoj technologií.

Snahou této práce je ukázat aproximaci Taylorovým polynomem jako dostatečně rychlou a přesnou metodu pro reálné použití. Práce je rozdělena na dvě části, teoretickou a praktickou. V teoretické částí je ukázána možnost aproximace integrálu pomocí libovolné aproximační metody. Následně se práce zabývá využitím Taylorova polynomu k aproximaci vícerozměrných integrálů a zvýšeni přesnosti integrování pomocí víceslovní aritmetiky. Pokud zvýšíme přesnost nad požadovanou mez, můžeme následně snížit počet kroků a tím dosáhnout stejně přesných výsledků s nižšími nároky na strojový čas. V poslední části se práce věnuje urychlením výpočtu numerického integrálu pomocí aproximace Taylorovým polynomem pomocí masivního paralelizmu. Dnes existují dvě hlavní platformy pro masivní

paralelizmus reprogramovatelný hardware (FPGA) a obecné výpočty prováděné na grafických kartách (GPGPU).

Zajímavou literaturou je [10], zde je zmíněno několik zajímavých algoritmů. Algoritmus převodu Lagrangeova interpolačního polynomu na Taylorův polynom a algoritmus výpočetu inverzní matice. V knize [3] se nachází několik důležitých definic. Rozsáhlou publikací o aproximováni funkcí je kniha [7].

Kapitola 2

Integrál

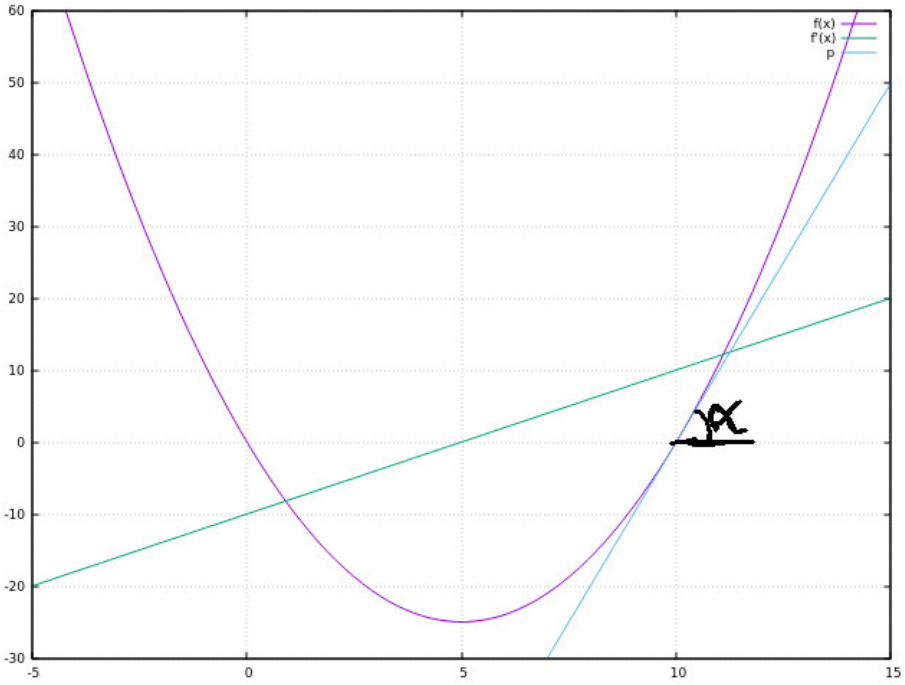
K dobrému pochopení integrálního počtu je nutné znát derivace, limity a geometrii. I když výpočty kvadratur jsou známé od starověku, vznik integrálního počtu je spjat s modernější historií matematiky, zejména jsou s ním spojeni dva vědci: Issac Newton (1643-1727) a Gottfried Wilhelm Leibniz(1646-1716). Pomocí integrálního počtu lze počítat různé obsahy a objemy těles ohraničených křivkami, rovinami případně jinými vícerozměrnými tělesy, například hyperkrychlí. Pomocí integrálního počtu lze také odvodit různé vzorce pro výpočet obsahu specifičtějších, ale přesto ještě obecných těles. Klasickým odvozením bývá obsah koule.

Začněme s jednoduchým vysvětlení derivací: Mějme funkci $y = f(x)$ definovanou na množině $M =< a; b >$ k níž sestrojíme funkci $f'(x)$ definovanou na množině M. Sestrojenou funkci $f'(x)$ nazvěme derivací funkce $f(x)$. Definujme přímku p, která je tečnou funkce $f(x)$ v bodě x_n. Přímka p svírá s osou x úhel α, pak pro úhel α platí rovnost $f'(x_n) = \tan(\alpha)$. Na obrázku 2.1 je zobrazena funkce $f(x) = x^2 - 10x$ a její derivace $f'(x) = 2x - 10$. Přímka $p(x) = 10x - 100$ představuje tečnu funkce $f(x)$ v bodě $x = 10$.

Z matematického hlediska lze derivaci také vyjádřit pomocí vzorce využívající limity.

$$f'(x) = \lim_{\Delta x -> 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Ze vzorce je patrné, že derivaci $f'(x)$ můžeme také chápat jako změnou funkce $f(x)$ na nekonečně malém intervalu v daném bodě x. Úhel α můžeme také reprezentovat jako vektor. Tento vektor lze v grafu zobrazit nakreslením vektoru $(1; f'(x))$. Pokud hodnota funkce $f'(x)$ je kladná, pak funkce $f(x)$ je funkcí rostoucí v daném bodě x. Naopak pokud hodnota funkce $f'(x)$ je záporná, potom je funkce $f(x)$ klesající v daném bodě x. Číslo nula není ani kladné, ani záporné. V tomto případě se tedy nejedná ani o funkci klesající, ani o funkci rostoucí.



Obrázek 2.1: Zobrazení funkcí f(x), f'(x) a přímky p pro $x_n = 10$

2.1 Neurčitý integrál

Úloha spočívá v nalezení takové funkce $F(x)$ k funkci $f(x)$, která by splňovala podmínku $F'(x) = f(x)$. Pro neurčitý integrál platí, že integrál funkce je zase funkce. Pro složitější integrály je možné použít například tabulky [12].

2.2 Určitý integrál

Pokud se bude jednat o jednorozměrný integrál, pak po dosazení konkrétní hodnoty a za proměnnou x dostaneme obsah plochy. Obsah plochy je omezen křivkou $f(x)$, osou x, osou y a rovnoběžkou s osou y protínající bod a na ose x. Pak pro výpočet obsahu plochy omezeného křivkou $f(x)$, osou x, a dvěma rovnoběžkami s osou y a protínajícími body a, b plati:

$$I = \int_a^b f(x)dx = \int_J f(x)dx = [F(x)]_a^b = F(b) - F(a)$$

Funkce $f(x)$ musí být spojitá na intervalu $J =< a; b >$ Pro úplnost uveďme Weierstrassovu větu, pomocí které budeme následně definovat Riemanův určitý integrál. Weierstrassova věta říká: Pokud funkce $f(x)$ je spojitá na omezeném a uzavřeném intervalu $< a; b >$. Pak je funkce $f(x)$ na intervalu $< a; b >$ omezená a nabývá na něm svého minima i maxima. Tedy v intervalu $< a; b >$ existují body x_m a x_M pro které platí:

$$m = f(x_m) = \min f(x)|x \in < a; b >$$

$$M = f(x_M) = \max f(x)|x \in < a; b >$$

[5, str. 35]

Určitý Riemanův integrál je definovaný pomocí horních a dolních součtů. Budiž tedy dán interval $< a; b >$ na kterém chceme integrovat. Budiž také dána funkce $f(x)$ omezena na intervalu $< a; b >$, kterou chceme integrovat. Nyní rozdělme interval $< a; b >$ na n stejných částí. Dostaneme tedy $n + 1$ bodů $x_0, x_1, ..., x_n$ jenž splňují vztahy.

$$a = x_0 < x_1 < ... < x_{n-1} < x_n = b$$

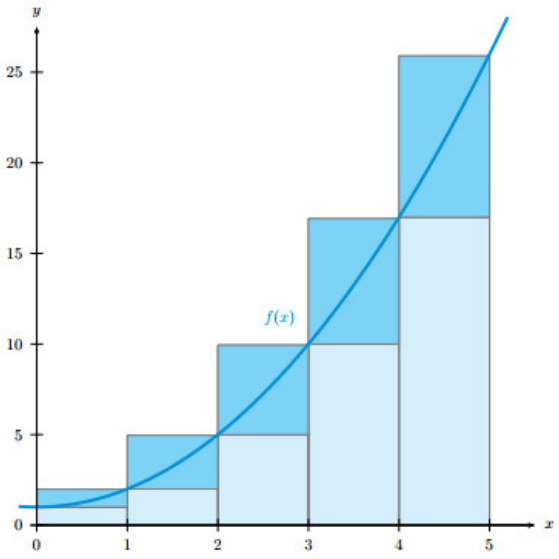
Tyto body dělí interval $< a; b >$ na n částečných intervalů $< x_0; x_1 >, < x_1; x_2 >, ..., < x_{n-1}; x_n >$, a nazýváme je dělícími body. Toto rozdělení, definované dělícími body $x_0, x_1, ..., x_n$ označme písmenem D. Délku částečného intervalu $< x_{i-1}, x_i >$ označme Δ_i, tj. položme $\Delta_i = x_i - x_{i-1}$. Následně označme znakem M_i supremum a znakem m_i infium funkce $f(x)$ v intervalu $< x_{i-1}; x_i >$. Danému rozdělení D přiřadíme nyní dvě čísla: číslo

$$S(D) = \sum_{i=1}^{n} M_i * \Delta x_i$$

jenž budeme nazývat horním součtem příslušným k rozdělení D, a číslo

$$s(D) = \sum_{i=1}^{n} m_i * \Delta x_i$$

jenž budeme nazývat dolním součtem příslušným k rozdělení D. Pro názornost slouží obrázek 2.2 zobrazující horní a dolní Riemanův integrál. Tmavě modře je zobrazen horní součet Riemanova integrálu a světle modře je zobrazen dolní součet Riemanova integrálu.



Obrázek 2.2: Dolní a horní Riemanův určitý integrál [9]

Z předchozí definice je jasné, že platí $m_i \leq M_i$. Tudíž taky platí $s(D) \leq S(D)$. Nyní definujme rozdělení D', které obsahuje všechny dělící body obsažené v rozdělení D. Nazvěme rozdělení D' zjemněním rozdělení D. Pak pro tyto dvě rozdělení platí:

$$D \subset D'$$

$$s(D) \leq s(D') \leq (S(D')) \leq S(D)$$

Následně pro rozdělení D^n u kterého délka částečného intervalu Δ_i se nekonečně blíží k nule, tedy $\Delta_i = \lim_{x_{i-1}->x_i} x_i - x_{i-1}$ platí.

$$s(D^n) = S(D^n)$$

$$\int_K f(x)dx = s(K) = S(K)$$

Z této definice je patrné, proč je možné rozdělit výpočet jednorozměrného integrálu na součet dvou jednorozměrných integrálů.

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

Jelikož i integrál funkce je funkce, lze tuto definici rozšířit i na dvojrozměrný integrál

$$\int_a^b \int_e^d f(x,y)dydx = \int_a^c \int_e^d f(x,y)dydx + \int_c^b \int_e^d f(x,y)dydx$$

Definice Reimanova integrálu s pomocí Weierstrassovy věty neumožňuje počítat integrály funkce $f(x)$ na intervalu $< a, b >$. Kde bod c leží v intervalu $< a, b >$ a funkce $f(x)$ je v bodě c nedefinovaná. Takovýto integrál lze vypočítat pomocí limity.

$$\int_a^b f(x)dx = (\lim_{x->c} F(x) - F(a)) + (F(b) - \lim_{x->c} F(x))$$

2.3 Vícerozměrný integrál

Pro jednoduchost nejprve definujme dvojrozměrný integrál pomocí substituce. Pro definici je důležité definovat jednorozměrnou integraci funkce více proměných. Zde se chováme k proměné, podle které se integruje, normálně a k ostatním proměným se chováme jako ke konstantě. Jak je uvedeno výše integrál funkce je zase funkce. Můžeme dvojrozměrný integrál definovat jako jednorozměrný integrál funkce, která je zase integrálem. Tedy integrál I lze vypočítat jako:

$$f(x,y) = \int g(x,y)dy$$

$$I = \int f(x,y)dx = \int \int g(x,y)dydx$$

Dvojrozměrný integrál můžeme také definovat podobně jako Riemanův integrál. Definujme tedy rozdělení $D =< a; b > \times < c; d >$ jako rozdělení plochy. Příklad rozdělení plochy je zobrazen na obrázku 2.3. Nyní definujme rozdělení plochy D' jako zjemnění rozdělení plochy D. Pro toto zjemnění bude také platit, že všechny body obsažené v rozdělení plochy D jsou také v zjemněném rozdělení plochy D', tedy $D \in D'$. Nyní mějme funkci $f(x,y)$, která je spojitá na intervalu $A = A_0 \times A_1$, kde A_0 a A_1 jsou jednorozměrné intervaly. Funkce $f(x,y)$ musí na intervalu A nabývat svého maxima a minima.

$$m = f(x_m) = \min f(x,y) | x \in A_0 \wedge y \in A_1$$

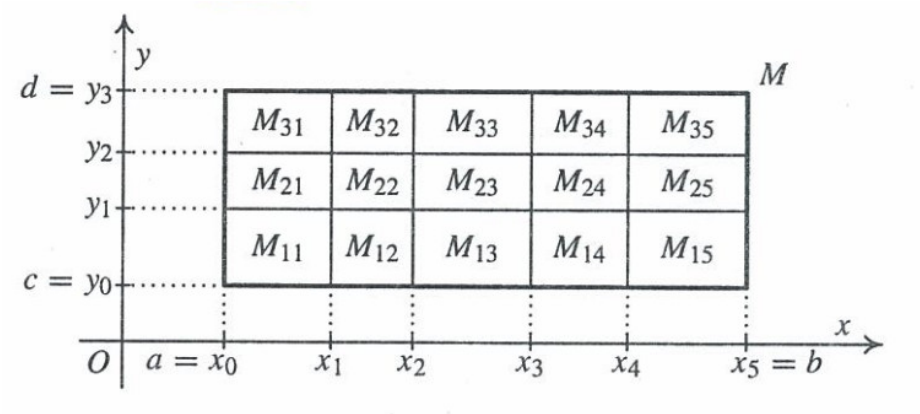
$$M = f(x_M) = \max f(x,y) | x \in A_0 \wedge y \in A_1$$

Definice je velice podobná definici jednorozměrného integrálu, jen se změnila množina skaláru na množinu vektorů. Tuto množinu můžeme také uspořádat, jen je nutné definovat řazení prvků. Například nejdříve seřadíme prvky podle hodnoty x. Prvky které mají stejnou hodnotu x následně seřadíme podle hodnoty y. Nyní můžeme definovat horní a dolní součet.

$$s(D) = \sum_{i=1}^{n}\sum_{j=1}^{m} m_{ij} * \Delta_{ij}$$

$$S(D) = \sum_{i=1}^{n}\sum_{j=1}^{m} M_{ij} * \Delta_{ij}$$

Pokud nyní dělení dostatečně zjemníme, pak $\Delta_{ij} = (\lim_{x_{i-1}->x_i} x_i - x_{i-1}) * (\lim_{y_{j-1}->y_j} y_j - y_{j-1})$ bude zde platit námi již známý vztah $S(D^n) = s(D^n)$. Pro srovnání udejme, že jednorozměrný integrál je součet nekonečně úzkých obdélníků. A dvojrozměrný integrál je součet kvádrů o nekonečně malé podstavě.



Obrázek 2.3: Rozdělení dvourozměrného integrálu [6]

Nyní definujme obecný N-rozměrný integrál. N-rozměrným intervalem budeme rozumět množinu, která je kartézským součinem jednorozměrných intervalů. Tedy $A = A_0 \times A_1 \times ... \times A_n$. Množiny A_i mohou být omezené, neomezené, uzavřené, otevřené, nebo polootevřené [6]. Nechť dělení $D_i =< x_{i0}; x_{i1}; ...; x_{in} >$ je dělení množiny A_i. Označme $D = D_0 \times D_1 \times ... \times D_n$ dělení n-rozměrného intervalu. Obrázek 2.3 zobrazuje dělení dvojrozměrného intervalu A. Definujme zjemnění dělení intervalu $D^{i-1} \subset D^i$. Pro vyjádření Δ_a použijeme Euklidovu normu.

$$\Delta_{ik...j} = max\sqrt{(x_i - x_{i-1})^2 + (y_k - y_{k-1})^2 + ... + (z_j - z_{j-1})^2}$$

kde $x_i \in D_0 \wedge y_k \in D_1 \wedge .... \wedge z_j \in D_n$.

Buď $f(\vec{x})$ ohraničená funkce n-proměnných definovaná na množině A. Nechť D je dělení množiny A. Definujme horní $S(M)$ a dolní $s(M)$ součet jako:

$$S(D) = \sum_{i=1}^{m}\sum_{j=1}^{n}...\sum_{z=1}^{u} m_{ij...z} * \Delta_{ij...z}$$
$$s(D) = \sum_{i=1}^{m}\sum_{j=1}^{n}...\sum_{z=1}^{u} M_{ij...z} * \Delta_{ij...z}$$

Následně pokud $\lim \Delta_{ik...j} = 0$ platí pro dělení D^n, potom také platí

$$s(D^n) = S(D^n) = \int \cdots \int_A f(\vec{x}) dx_0 ... dx_n$$

Pro integrování vícerozměrných integrálů existuje Fubiniova věta [6], která zjednodušuje integrování vícerozměrných integrálu záměnou pořadí integrace proměných.

$$\iint_M f(x,y)dxdy = \int_a^b \left[\int_c^d f(x,y) \right] dx, dy = \int_c^d \left[\int_a^b f(x,y) \right] dy, dx$$

Kde $M =< a; b > \times < c; d >$. I když je možné integrovat v libovolném pořadí. Bývá jedna varianta výhodnější. Nebo pokud by jedna z mezí c, d byla funkce závislá na nějaké proměnné, musel by být integrál vypočítán v určitém pořadí. Například:

$$\int_a^b \left[\int_{g_1(y)}^{g_2(y)} f(x,y) \right] dx, dy$$

2.4 Transformace integrálu

Transformace integrálu je velice důležitou součástí integrálního počtu. Pomocí transformace lze zjednodušit integrál převodem na jiný integrál. Jednoduše řečeno transformace je zdeformování prostoru ve kterém je integrál počítán. Deformace prostoru se řídí podle námi definované obecné funkce. Mezi nejčastější transformace integrálu patří transformace do válcových souřadnic, translace (posunutí), dilatace, transformace do sférických souřadnic.

Transformace jednoduchého integrálu se na středních školách často skrývá pod názvem substituční metoda.

$$\int_\alpha^\beta f(x)dx = \int_{\varphi(\alpha)}^{\varphi(\beta)} f(\varphi(t)) * \varphi'(t)dt$$

Kde $f(x)$ je spojitá funkce na intervalu $< a, b >$ a $\varphi(t)$ je spojitá funkce na intervalu $< \varphi(\alpha); \varphi(\beta) >$, přičemž $t \in < a; b >$ pro $\forall \varphi(t) \in < \alpha; \beta >$. V této metodě jde o zjednodušení integrálu do lépe spočitatelné podoby.

Pro obecný tedy n-rozměrný integrál platí:

$$\int_A \cdots \int f(\vec{x})d\vec{x} = \int_B \cdots \int f(g_1(\vec{y}), g_2(\vec{y}), ...g_n(\vec{y})) * j(\vec{y})d\vec{y}$$

Následně je potřeba vypočítat Jakobián J [6]:

$$j(\vec{y}) = J = \begin{vmatrix} \frac{\partial g_1(\vec{y})}{\partial y_1} & \frac{\partial g_1(\vec{y})}{\partial y_2} & \cdots & \frac{\partial g_1(\vec{y})}{\partial y_n} \\ \frac{\partial g_2(\vec{y})}{\partial y_1} & \frac{\partial g_2(\vec{y})}{\partial y_2} & \cdots & \frac{\partial g_2(\vec{y})}{\partial y_n} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial g_n(\vec{y})}{\partial y_1} & \frac{\partial g_n(\vec{y})}{\partial y_2} & \cdots & \frac{\partial g_n(\vec{y})}{\partial y_n} \end{vmatrix}$$

Kde $\vec{x} \in R^n$ a $\vec{y} \in R^n$. Funkce $f(\vec{x})$ je spojitá na množině A. Dále $\forall i : g_i(\vec{y})$ jsou spojité na množině B, přičemž platí $\forall i : g_i(\vec{y}) \in B$ pro $\forall \vec{y} \in A$. Jakobián se nesmí rovnat nule $j(\vec{y}) = J \neq 0$ pro $\forall \vec{y} \in B$.

Uveďme příklad z literatury [6].Vypočítejme integrál, kde množina A je omezena křivkami $y = 1/x; y = 3/x; y = x/2, y = 2x$. Transformací integrálu se značným způsobem zjednoduší výpočet.

$$\iint\limits_A f(x,y)dxdy = \iint\limits_B f(g(u,v),h(u,v))J(u,v)dudv$$

$$J(u,v) = \begin{vmatrix} g'_u(u,v) & g'_v(u,v) \\ h'_u(u,v) & h'_v(u,v) \end{vmatrix} \neq 0$$

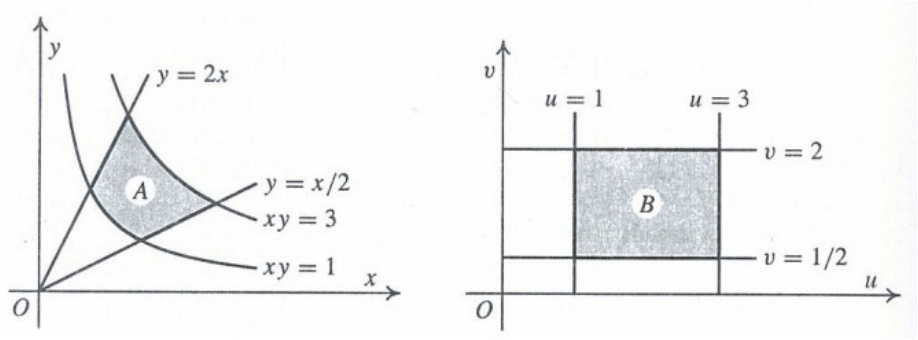
Omezení jsou dvě hyperboly a dvě přímky. Každým vnitřním bodem prvního kvadrantu s kartézskými souřadnicemi $[x_0; y_0]$ prochází právě jedna z hyperbol $y = u_0/x$ a právě jedna z přímek $y = v_0 x$, kde $u_0 > 0$, $v_0 > 0$ jsou parametry. Čísla u_0 a v_0 jsou jednoznačně určena: $u_0 = x_0 y_0$ a $v_0 = y_0/x_0$. Dvojici $[u_0, v_0]$ lze tedy zvolit za nové souřadnice daného bodu. Vztah mezi původními a novými souřadnicemi je tudíž dán rovnicemi $xy = u$ a $y/x = v$. Z nich snadno vypočítáme $x = \sqrt{u/v}$, $y = \sqrt{uv}$. Dostáváme tedy prosté zobrazení se souřadnicovými funkcemi $g(u,v) = \sqrt{u/v}$ a $h(u,v) = \sqrt{uv}$. Tyto funkce mají uvnitř prvního kvadrantu spojité první derivace podle obou proměnných. Jákobián je:

$$J(u,v) = \begin{vmatrix} \frac{1}{2}\sqrt{\frac{1}{uv}} & \frac{-1}{2}\sqrt{\frac{u}{v^3}} \\ \frac{1}{2}\sqrt{\frac{u}{v}} & \frac{1}{2}\sqrt{\frac{u}{v}} \end{vmatrix} = \frac{1}{2v}$$

Body ležící na hyperbole $xy = 1$ leží na v prostoru B na přímce $u = 1$ analogicky body hyperboly $xy = 3$ leží v prostoru B na přímce $u = 3$. Podobně body ležící na přímce $y = x/2$ leží nyní na přímce $v = 1/2$ a body na přímce $y = 2x$ nyní všechny leží na přímce $v = 2$

$$\iint\limits_A 1dydy = \iint\limits_B \frac{1}{2v}dudv = \frac{1}{2} * \left[\ln(v) * [u]_1^3\right]_{\frac{1}{2}}^2 = 2\ln(2)$$

Na obrázku 2.4 je vidět vzhled množin A, B po transformování integrálu.



Obrázek 2.4: Transformace integrálu z množina A do množiny B [6]

Kapitola 3

Numerická integrace

Pro výpočetní techniku jsou daleko přirozenější numerické výpočty než analytické. Numerickými metodami lze počítat pouze určité integrály. Metody numerické integrace můžeme rozdělit na dva druhy metod.

Metody prvního druhu numerické integrace vycházejí z podstaty aproximace funkce jinou funkcí. Definujme funkci $f(x)$ a aproximaci $A(x)$ této funkce. Pokud tedy aproximační funkci $A(x)$ integrujeme dostaneme aproximaci integrálu $I = \int A(x)dx$ dané funkce. Metoda využívá vlastnosti určitých integrálů, kdy určitý integrál dvou funkcí, které mají na integrovaném intervalu stejný průběh je stejný. Nechť je dána funkce $y = f(x)$ a je třeba vypočítat integrál

$$\int_a^b f(x)dx$$

Nyní lze nahradit funkci $f(x)$ aproximační $A(x)$ funkcí, kde $R(x)$ je chyba aproximace. Pak dostaneme rovnici:

$$\int_a^b f(x)dx = \int_a^b (A(x) + R(x))dx \approx \int_a^b A(x)dx$$

Tyto metody neintegrují samotnou funkci $f(x)$, ale aproximaci $A(x)$ dané funkce, která je velice podobná funkci $f(x)$. Mezi tyto metody patří Newtonovy-cotesovy kvadraturní formule, Kubaturní formule a jiné.

Metody druhého druhu převádí integrál na diferenční rovnici s počáteční podmínkou $y(a) = 0$ pro počáteční bod a. Derivace funkce udává její průběh. Výpočet integrálu funkce $f(x)$ můžeme zapsat:

$$F(x) = \int f(x)dx$$

Pokud se rovnice zderivuje vznikne diferenciální rovnice (3.1). Nyní známe derivaci funkce, tedy i její průběh v jednotlivých bodech x. Počítače jsou však diskrétní systémy a výpočet reálných problémů je nutné vždy převést na diskrétní problémy.

$$F'_x(x) = f(x) \tag{3.1}$$

Přesná aproximace funkce na celém intervalu $< a; b >$ je velice nepřesná, nebo velice náročná. Proto je potřeba výpočet numerického integrálu rozdělit na n částí. Tímto rozdělením dojde k výraznému zpřesnění výpočtu. Mějme množinu $A =< a; b >$ na které chceme funkci $f(x)$ numericky integrovat, kde funkce $f(x)$ je na intervalu A spojitá. Definujme uspořádanou množinu $B = \{x_0; x_1; ...; x_n = b\} \subset A$, kde $a = x_0 < x_1 < ... < x_n = b$.

Množina B je zjemněním dělení množiny A. Integrál se vypočítá jako součet všech integrálů omezených dvěma vedle sebe ležícími body v množině B. Pro lichoběžníkové pravidlo můžeme využít vzorec níže zobrazený.

$$\int_a^b f(x) = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x) \approx \sum_{i=1}^{n} (x_i - x_{i-1}) * \frac{f(x_i) + f(x_{i-1})}{2}$$

$$x \in B$$

3.1 Vícerozměrný integrál

Metody lze použít pro výpočet dvou a více rozměrných integrálu a to dvěma způsoby. První jednodušší způsob je nejdříve aproximovat integrál prvního řádu. Následně výsledky integrace prvního řádu použít pro výpočet integrálu druhého řádu.

$$F(x,y) = \iint f(x,y)dxdy$$

$$F'_x(x,y) = \int f(x,y)dy$$

$$F''_{xy}(x,y) = f(x,y)$$

Řešení diferenciální rovnice vyššího řádu lze rozdělit na řešení dvou diferenciálních rovnic prvního řádu.

$$F'_x(x,y) = G(x,y)$$

$$G'_y(x,y) = f(x,y)$$

Pro řešení vícerozměrného integrálu lze využít aproximačních funkcí pro aproximaci ploch. Tedy vícerozměrný Taylorův polynom nebo kubaturní formule [2, p683]. Rozměr integrálu není nikdy omezen, proto je nutné počítat vícerozměrné integrály metodami, kterými se počítají méněrozměrné integrály.

$$\iint_D f(y,x)dxdy = \int_a^b \int_c^d f(y,x)dxdy = \int_a^b F(y)dy$$

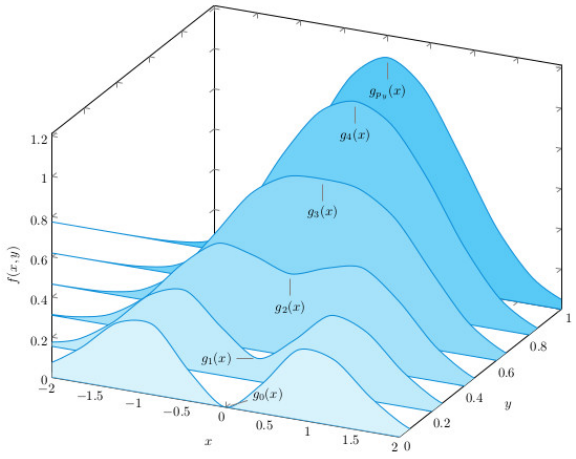
Kde

$$D =< a; b > \times < c; d >$$

$$\int_c^d f(y,x)dx = F(y)$$

Vícerozměrný integrál je převeden na jednorozměrný integrál, přitom vnitřní integrál je chápán jako funkce. Pro ukázku uveďme numerický výpočet výše uvedeného obecného dvojrozměrného integrálu. Nejprve rozdělíme interval $< a; b >$ na více intervalů s dělícími body $y_0, y_1, ..., y_{n-1}, y_n$. Pro jednotlivé dělící body spočteme hodnoty funkce $F(y_0), F(y_1), ..., F(y_{n-1})$. Hodnoty v těchto bodech se využijí k výpočtu integrálu vyššího řádu. Například proložením polynomu přes výsledné hodnoty získáme aproximaci neurčitého integrálu $G(x) = \int F(x)$.

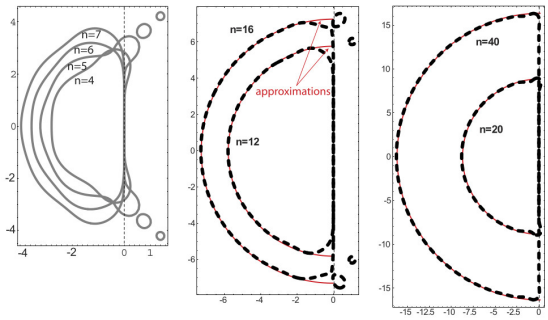
$$
\begin{aligned}
F(y_0) &= \int_a^b f(y_0, x)dx \\
F(y_1) &= \int_a^b f(y_1, x)dx \\
&\vdots \\
F(y_{n-1}) &= \int_a^b f(y_{n-1}, x)dx
\end{aligned}
\tag{3.2}
$$



Obrázek 3.1: Dvojrozměrný integrál [9]

3.2 Oblast stability

Numerické metody pro řešení počátečních úloh mají oblast stability. Metodu je nutné pří výpočtu držet v oblasti stability, jinak by bylo výsledkem nepřesné a nepoužitelné řešení. Stabilita je většinou zobrazována na grafu s reálnou a imaginární osou. Na takovémto grafu lze zobrazit různé systémy, které aproximujeme. Graf lze rozdělit na dvě půlky. Systémy, které se nachází v kladné části reálné osy, jsou nestabilní. Systémy, které se nachází v záporné části reálné osy, jsou stabilní. Přičemž oba systémy lze přesně aproximovat. Pro řešení určitých problémů je nutné zjistit oblast stability a zvolit patřičnou metodu k problému. Nestabilní systémy lze řešit některými numerickými metodami. Jejich řešení sice bude správné, ale bude odpovídat nestabilnímu systému. Některé metody mají oblast stability celou zápornou část komplexní poloroviny. Nazýváme je A-stabilními metodami.



Obrázek 3.2: oblast stability některých metod [1]

Zajímavým problémem z pohledu stability metod je Dahlquistův problém [18]. Mějme diferenciální rovnice ve tvaru:

$$y' = \lambda y, y(0) = 1, \lambda < 0$$

Analytické řešení problému je ve tvaru

$$y(x) = y(0)e^{\lambda x}$$

Dahlquistův problém se s rostoucí hodnotou konstanty $|\lambda|$ dostává mimo oblast stability většiny numerických metod.

3.3 Newtonovy-cotesovy kvadraturní formule

Pro výpočet integrálu uvažujme funkci $y = f(x)$ a počítejme integrál

$$\int_a^b f(x)$$

Rozdělme interval $< a; b >$ pomocí ekvidistantní sítě na n stejných částí. Lze použít i neekvidistantní síť, ale výpočet koeficientů je daleko složitější díky nepravidelnému umístění bodů. Použitím neekvidistantní sítě lze zpřesnit aproximaci některých funkcí[7], tím pádem i aproximaci hledaného integrálu.

Pro účel výpočtu integrálu nahradíme funkci $f(x)$ Lagrangeovým polynomem n-tého řádu $L_n(x)$ aproximující danou funkci na intervalu $< a; b >$

$$\int_a^b f(x)dx \approx \int_a^b L_n(x)dx$$

Nyní lze integrovat samotný Langrangeův polynom. Podle pravidel přepíšeme integrál sumy na sumu integrálu. Jelikož jsou hodnoty x_i konstanty, potom je i funkce $f(x_i)$ v tomto bodě konstanta a tudíž integrace těchto hodnot je triviální.

$$\int_a^b L_n(x)dx = \int_a^b \sum_{i=0}^{n}(f(x_i) * l_i(x))dx = \sum_{i=0}^{n}(f(x_i) * \int_a^b (l_i(x))dx) \tag{3.3}$$

Funkce l_i je definována jako:

$$l_i(x) = \prod_{j=0, i\neq j}^{n} \frac{(x - x_j)}{x_i - x_j} \Rightarrow \int_a^b l_i(x)dx = \int_a^b \prod_{j=0, i\neq j}^{n} \frac{(x - x_j)}{x_i - x_j}dx = \prod_{j=0, i\neq j}^{n} \frac{1}{x_i - x_j} * \int_a^b \prod_{j=0, i\neq j}^{n} (x - x_j)dx$$

Pomocí Newton-cotestovy kvadraturní formule je možné odvodit Lichoběžníkovou a Simpsonovu formuli pro výpočet integrálu. Odvození Simpsonovy metody je uvedeno v literatuře [2]. Následující příklad zobrazuje odvození Lichoběžníkové metody. Jak je vidět ve vzorci (3.3) je pouze potřeba integrovat polynomy l_i, vše ostatní jsou konstanty. Za krajní body $a; b$ aproximace integrálu lze zvolit jiné body než $x_0; x_1$. U této metody to není příliš praktické, ale u metod odvozených z polynomů vyšších řádů se to může hodit. Výsledkem by byla interpolace integrálů pomocí vnějších bodů. Pro Langrangeův polynom druhého stupně L_2 platí:

$$\int_a^b l_i(x)dx$$

$$\int_a^b l_0(x)dx = \int_a^b (\frac{x - x_1}{x_0 - x_1})dx = \frac{1}{x_0 - x_1}\int_a^b (x - x_1)dx = \left[\frac{x^2 - 2x * x_1}{2(x_0 - x_1)}\right]_a^b$$

$$\int_a^b l_1(x)dx = \int_a^b (\frac{x - x_0}{x_1 - x_0})dx = \frac{1}{x_1 - x_0}\int_a^b (x - x_0)dx = \left[\frac{x^2 - 2x * x_0}{2(x_1 - x_0)}\right]_a^b$$

Dosazením x_0, x_1 za meze a, b dostaneme výsledek.

$$\int_{x_0}^{x_1} l_0(x)dx = \left[\frac{x^2 - 2x * x_1}{2(x_0 - x_1)}\right]_{x_0}^{x_1} = \frac{-(x_1 - x_0)^2}{2(x_0 - x_1)} = \frac{x_1 - x_0}{2}$$

$$\int_{x_0}^{x_1} l_1(x)dx = \left[\frac{x^2 - 2x * x_1}{2(x_1 - x_0)}\right]_{x_0}^{x_1} = \frac{(x_1 - x_0)^2}{2(x_1 - x_0)} = \frac{x_1 - x_0}{2}$$

Následně dostaneme vzorec

$$\int_{x_0}^{x_1} f(x)dx = \int_{x_0}^{x_1} L_2(x)dx = \frac{f(x_1) + f(x_0)}{2} * (x_1 - x_0)$$

3.3.1 Kubaturní formule

Kubaturní formule jsou určeny k numerickému výpočtu dvojných integrálů. Vznikly rozšířením Newtonovy-cotesovy kvadraturní formule do dvojrozměrného prostoru. Nechť $z = f(x, y)$ je definovaná a spojitá v jisté omezené oblasti. Jak víme pro jednorozměrný integrál $F(x)$ platí.

$$F = \int_a^b f(x)dx = \sum_{i=1}^{N} A_i * f(x_i)$$

Pak pro dvojrozměrný integrál I musí platit:

$$I = \int_a^b \int_c^d f(x, y)dxdy = \int_a^b F(y)dy = \sum_{i=1}^{N} A_i F(y_i)$$

Nyní aproximujme integrál $F(x)$:

$$I = \int_a^b F(y)dy = \sum_{i=1}^{N} A_i F(y_i) = \sum_{i=1}^{N} A_i * \left(\sum_{j=1}^{M} B_j * f(x_i, y_j)\right)$$

$$I = \sum_{i=1}^{N} \sum_{j=1}^{M} A_i * B_j * f(x_i, y_j)$$

Z obecného popisu kubaturní formule vyplývá následující vzorec pro výpočet integrálu I kubaturní formulí simpsonova typu. Odvození udělejme postupně. Nejdříve vyjádříme dvojrozměrný integrál jako jednorozměrný integrál počítaný simpsonovou metodou.

$$I = \int_a^b \int_c^d f(x,y)dxdy = \sum_{i=1}^{3} A_i * F(x_i) = \frac{h}{3} * (F(x_1) + 4 * F(x_2) + F(x_3))$$

Nyní přidáme výpočet vnitřního integrálu.

$$I = \int_a^b \int_c^d f(x,y)dxdy = \frac{h}{3} * (\sum_{i=1}^{3} B_i * f(x_1, y_i) + 4 * \sum_{i=1}^{3} B_i * f(x_2, y_i) + \sum_{i=1}^{3} B_i * f(x_3, y_i))$$

Z čehož vyplývá vzorec pro výpočet integrálu I kubaturní formulí simpsonova typu:

$$\begin{aligned} I = \int_a^b \int_c^d f(x,y)dxdy = \frac{h*k}{9} * \\ \{[f(x_1,y_1) + 4f(x_2,y_1) + f(x_3,y_1)] + \\ [4f(x_1,y_2) + 16f(x_2,y_2) + 4f(x_3,y_2)] + \\ [f(x_1,y_3) + 4f(x_2,y_3) + f(x_3,y_3)]\} \end{aligned} \tag{3.4}$$

3.4 Metoda Monte carlo

Klasickým metodám při zvětšujícím se počtu rozměrů narůstá exponenciálně složitost výpočtu. Metoda Monte carlo využívá statistiku a pravděpodobnost k zjednodušení a zpřesnění výpočtu. Příkladem buď výpočet m-rozměrného integrálu I:

$$I = \int_S \cdots \int f(x_1, x_2, ..., x_m)dx_1dx_2...dx_m$$

Geometricky představuje integrál $(m+1)$ rozměrné těleso s podstavou S. Podstavu S ohraničíme kvádr A tak, že platí $S \subset A$. Následně integrál I transformujeme tak, aby kvádr A se stal m rozměrnou jednotkovou krychlí. Množina S se také transponovala, a stále leží uvnitř jednotkové krychle A. Mějme množinu náhodných čísel M vygenerovanou algoritmem, které náleží do množiny A.

$$M \subseteq A \subset R^n$$

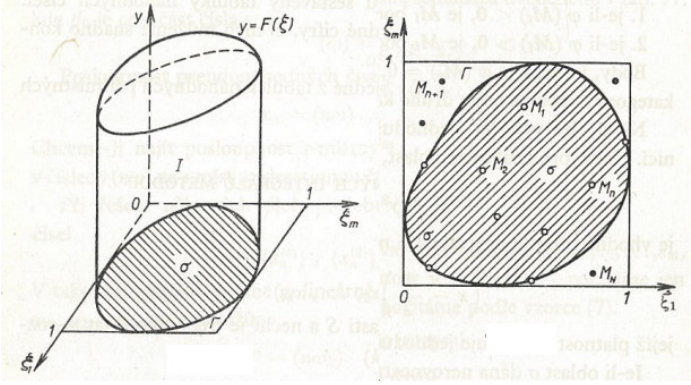
$$S \subseteq A \subset R^n$$

Do sumy se započítávají pouze prvky, které se nacházejí v množině S. Proto definujme množinu N, která bude obsahovat pouze prvky obsažené v S. Tedy x je náhodná hodnota vygenerovaná algoritmem patřící do množiny S. Pak pro integrál I platí:

$$N = \{x | x \in M \wedge x \in S\}$$

$$I = \frac{1}{|M|} * \sum_{i=1}^{|N|} f(N_i)$$

Větší přesnost integrálu lze dosáhnout zvýšením počtu prvků v množině A.



Obrázek 3.3: Podstava aproximace integrálu [2]

3.5 Metoda Runge-Kutta

Metody Runge-Kutta používají vážený průměr derivací v několika bodech mezi počátečním a koncovým bodem. Jsou jedněmi z nejpoužívanějších metod pro řešení počátečních úloh. Obecný tvar metod Runge-Kutta je zobrazen vzorcem (3.5).

$$y_n = y_{n-1} + h * \left(\sum_{i=1}^{m} w_i * k_i \right) \qquad \sum_{i=1}^{m} w_i = 1 \tag{3.5}$$

Kde m je řád metody a k_i je derivace vypočtená v i-tém řádu metody. Pro metody Runge-Kutta existují explicitní a implicitní varianty. Přičemž implicitní varianty mívají větší stabilitu. Metoda Runge-Kutta prvního řádu je shodná s Eulerovou metodou. Níže je zobrazen vzorec pro explicitní Eulerovu metodu (3.6) a implicitní Eulerovu metodu (3.7) a explicitní metodu Runge-Kutta čtvrtého řádu (3.8) se zadanou rovnicí $y' = f(x, y)$.

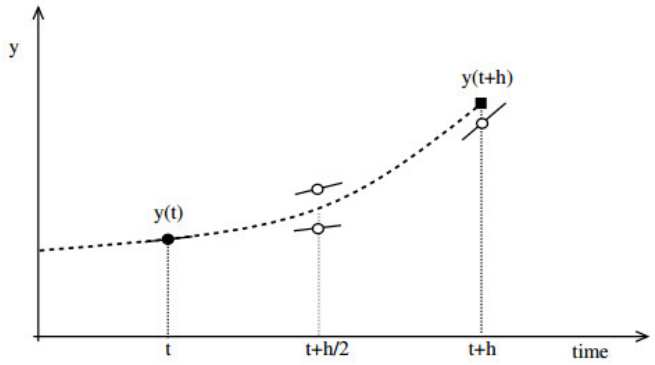
$$y_{n+1} = y_n + h * f(x_n, y_n) \tag{3.6}$$

$$y_{n+1} = y_n + h * f(x_{n+1}, y_{n+1}) \tag{3.7}$$

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(x_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + h(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}) \end{aligned} \tag{3.8}$$

Na obrázku 3.4 je zobrazen výpočet metody Runge-Kutta čtvrtého řádu. V prvním kroku je vypočítána derivace k_1 v bodě t. Pomocí derivace k_1 se následně posuneme do bodu $t + h/2$ a spočítáme derivaci k_2. Nyní se výpočet posune zpátky do bodu t. Následně se znovu posuneme do bodu $t + h/2$ nyní už ale s pomocí derivace k_2 a spočítáme derivaci

k_3. Pak pomocí derivace k_3 se posuneme z bodu t do bodu $t+h$ a zde je spočítána derivace k_4. Nakonec je udělán poslední posun z bodu t zprůměrováním derivací do bodu $t+h$



Obrázek 3.4: Výpočet metodou Runge-Kutta [14]

3.6 Metoda Taylorovy řady

Oproti ostatním metodám využívajícími pouze derivace prvního řádu metoda Taylorovy řady využívá derivace vyšších řádů v jednom bodě. Využitím vyššího počtu derivací je možné zvětšit krok simulace či integrace a přitom zachovat přesnost výpočtu. Bylo dokázáno, že metoda dosahuje velké přesnosti a rychlosti. Problémem je však výpočet derivací vyššího řádu. Ten lze provést několika způsoby. Jedním ze způsobů je využití inverzní matice vzniklé z Taylorovy řady. Tento způsob je popsán v kapitole 3.7. Dalším způsobem může být metodika tvořících diferenciálních rovnic. Tato metoda je nejpřesnější, ale zato složitější. Dále lze derivace určit numericky pomocí vzorce limity. Metoda výpočtu derivace pomocí limity je nejednodušší metodou, ale zato nejméně přesnou metodou výpočtu derivací.

$$f'(x) = \lim_{\Delta x -> 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

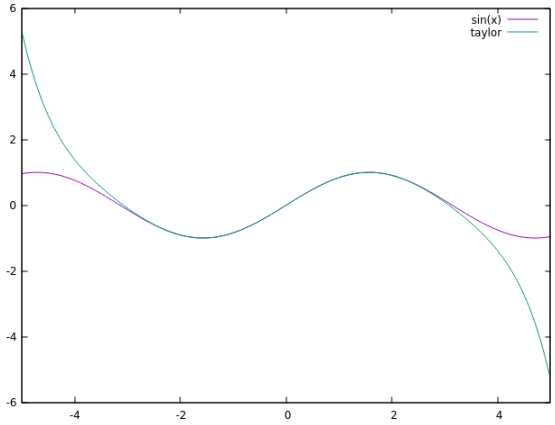
Tuto rovnici lze i derivovat a získat tím druhou, třetí, čtvrtou, atd derivaci.

$$f^{(n)}(x) = \lim_{\Delta x -> 0} \frac{f^{(n-1)}(x+\Delta x) - f(x)^{(n-1)}}{\Delta x}$$

Taylorův polynom je aproximační metodou funkce. Lze vyjádřit jak nekonečný součet derivací. Při využití všech derivací je aproximace absolutně přesná a popisuje přesně funkci $f(x)$.

$$f(x) = f(a) + \sum_{i=1}^{\infty} \left(\frac{f^{(i)}(a)}{i!} * (x-a)^i \right) = \sum_{i=0}^{\infty} \left(\frac{f^{(i)}(a)}{i!} * (x-a)^i \right) \tag{3.9}$$

Na obrázku 3.5 je zobrazena aproximace funkce $\sin(x)$ Taylorovým polynomem 7 řádu. Jelikož funkce $\sin(x)$ má nekonečně mnoho derivací, pro její přesnou aproximaci bychom museli využít nekonečného počtu derivací. Z obrázku je patrné, že aproximace funkce je relativně přesná od počátku na intervalu $< -2; 2 >$. Od jednotlivých bodů 4 a -4 začíná být aproximace velice nepřesná.



Obrázek 3.5: aproximace funkce $\sin(x)$ Taylorovým polynomem 7. řádu

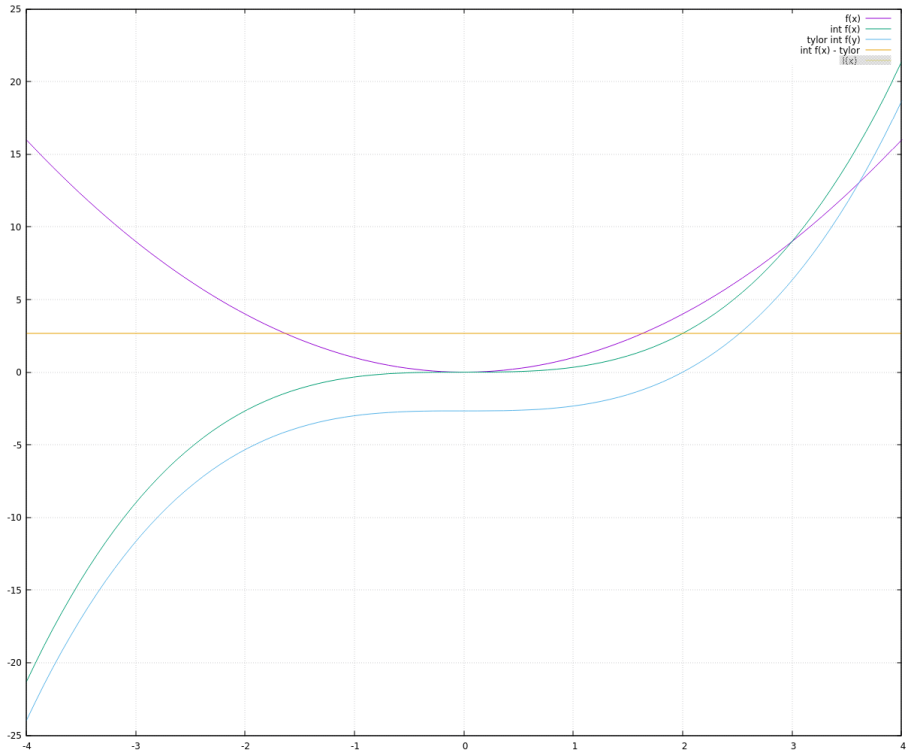
Aproximace určitého integrálu metodou Taylorovy řady předvedeme na příkladu obecného určitého integrálu $I = \int_a^b f(x)dx$. U výpočtu určitého integrálu metodou Taylorova polynomu můžeme psát $F(a) = 0$. Důkaz tohoto tvrzení budiž rovnice (3.10). Tento člen se ve výsledku vůbec neprojeví. Lze tedy použít jakoukoliv spodní mez a není vůbec nutné provádět transformaci, tak aby $a = 0$, jako je to navrhováno v některých publikacích.

$$\int_a^b f(x)dx = F(b) - F(a) = F(a) + \sum_{i=1}^{n} \frac{f^{(i)}(a)}{i!}(b-a)^i - F(a) = \sum_{i=1}^{n} \frac{f^{(i)}(a)}{i!}(b-a)^i \quad (3.10)$$

Výraz $F(b)$ je horní mez integrálu, pro kterou platí následující vzorec. V něm je zobrazen výpočet bodu $F(b)$ z bodu $F(a)$ pomocí metody Taylorova rozvoje.

$$F(b) = F(a) + \sum_{i=1}^{n} \frac{f^{(i)}(a)}{i!}(b-a)^i$$

Obrázek 3.6 zobrazuje aproximace určitého integrálu funkce $f(x) = x^2$ metodou Taylorova rozvoje s využitím maximálního počtu derivací. Celkem bylo využito třech derivací pro znázornění aproximační funkce určitého integrálu *taylor int* z bodu $x = 2$. Z obrázku to není přímo patrné ale výpočet integrálu pomocí metody Taylorovy řady s využitím všech derivací je stoprocentně přesný. Daný rozdíl mezi aproximací integrálu funkce Taylorovým polynomem *taylor int* a analytickým výpočtem integrálu *int* je konstantní hodnota rovna $\int f(2)$. K rozdílu došlo díky posunu začátku aproximace z bodu $x = 0$ do bodu $x = 2$. Analyticky lze dosáhnout stejného výsledku posunutím počátku integrálu do bodu $x = 0$. Tím získáme integrál $\int_2^y f(x)dx = F(y) - F(2)$.



Obrázek 3.6: Porovnání integrace Taylorovým polynomem z bodu $x = 2$

Počítače mají omezenou datovou přesnost a počítají v konečném čase. Proto není možné v metodě Taylorova polynomu využít nekonečně mnoho derivací. Aproximace na celém intervalu by tudíž byla neefektivní. S využitím definice Riemmanova určitého integrálu lze rozdělit výpočet integrálu I na množině J na součet několika určitých integrálů s menším krokem. Rozdělme tedy množinu J na n podmnožin. V každé podmnožině J_i vypočtěme určitý integrál I_i, pro které platí $J_i \cap J_k = \emptyset$ a $\cup J_i = J$. Výsledný integrál I získáme sečtením všech integrálů I_i. Ve vzorci (3.11) řádek znázorňuje výpočet integrálu. Výsledkem sečtením všech řádků je integrál I. Konstanta k představuje počet derivací použitých pro výpočet integrálu metodou Taylorova polynomu. Konstanta n je počet na kolik integrálů byl rozdělen výpočet. Konstanta C představuje počáteční hodnotu aproximace.

$$
\begin{aligned}
I_0 &= C \\
I_1 &= \sum_{i=1}^{k} \frac{f^{(i)}(x_0)}{i!} * (x_1 - x_0)^i \\
I_2 &= \sum_{i=1}^{k} \frac{f^{(i)}(x_1)}{i!} * (x_2 - x_1)^i \\
I_3 &= \sum_{i=1}^{k} \frac{f^{(i)}(x_2)}{i!} * (x_3 - x_2)^i \\
&\vdots \\
I_n &= \sum_{i=1}^{k} \frac{f^{(i)}(x_{n-1})}{i!} * (x_n - x_{n-1})^i \\
I &= \sum_{i=0}^{n} I_i
\end{aligned}
\tag{3.11}
$$

Díky vlastnosti sčítání lze použít integrál I_{i-1} jako počáteční hodnotu ve výpočtu následujícího integrálu I_i. Nyní nebudeme tedy počítat integrály na intervalech J_k, ale na intervalu $\cup J_k$, pro $k < i$.

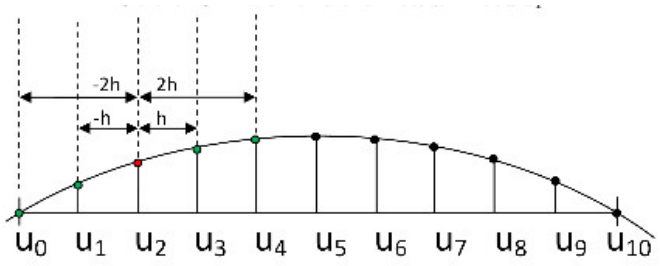
$$
\begin{aligned}
I_0 &= C \\
I_1 &= I_0 + \sum_{i=1}^{k} \frac{f^{(i)}(x_0)}{i!} * (x_1 - x_0)^i \\
I_2 &= I_1 + \sum_{i=1}^{k} \frac{f^{(i)}(x_1)}{i!} * (x_2 - x_1)^i \\
I_3 &= I_2 + \sum_{i=1}^{k} \frac{f^{(i)}(x_2)}{i!} * (x_3 - x_2)^i \\
&\vdots \\
I_n &= I_{n-1} + \sum_{i=1}^{k} \frac{f^{(i)}(x_{n-1})}{i!} * (x_n - x_{n-1})^i \\
I &= I_n
\end{aligned}
\tag{3.12}
$$

3.7 Numerický výpočet derivací pomocí diferencí

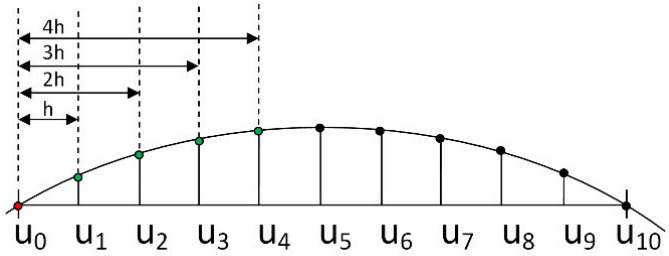
Při výpočtu určitých integrálů vyšších řádů je nutné využít numerických metod derivování. Pro numerické derivování, nebo taky aproximaci derivací vyšších řádů, je nutné vypočítat okolní body bodu, ve kterém chceme znát derivace vyšších řádů. Numerickým výpočtem derivací z okolních bodů se zabývá celá kapitola v literatuře [2]. Možnostmi jsou například Newtonova interpolační formule nebo Stirlingova interpolační formule. V této publikaci se budu zabývat aproximací derivací pomocí Taylorova polynomu uvedenou v práci[11].

Na obrázcích je zobrazena diference kombinovaná 3.7, diference dopředná 3.8, a diference zpětná 3.9. Všechny derivace jsou počítány z bodů $f(x)|x \in \{u_0, u_1, u_2, u_3, u_4\}$. Pro

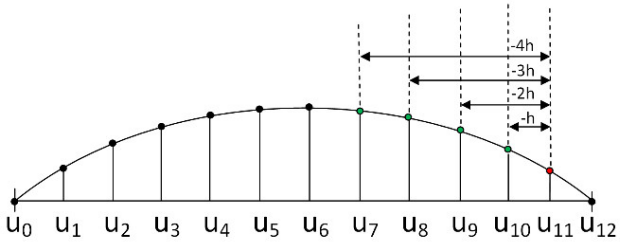
kombinovanou diferenci jsou derivace v bodě u_2. Pro dopřednou diferenci jsou počítána derivace pro bod u_0. Nakonec pro zpětnou diferenci jsou derivace počítány v bodě u_{11}



Obrázek 3.7: kombinovaná diference [9]



Obrázek 3.8: dopředná diference [9]



Obrázek 3.9: zpětná diference [9]

Pokud známe hodnoty derivací v bodě a funkce $f(x)$. Můžeme aproximovat hodnoty funkce $f(x)$ v okolí bodu a metodou Taylorova rozvoje. Obráceným postupem lze získat derivace v bodě a z hodnot funkce $f(x)$. Pokud tedy známe hodnoty funkce $f(x)$ v bodech $D = \{x_0, x_1, ..., x_n\}$. Můžeme aproximovat n derivací funkce $f(x)$. Sestavme tedy rovnice(3.13), kde x_k je bod pro který budou počítány derivace $f^{(i)}(x_k)$. Počet derivací, které chceme vypočítat je dán číslem n.

$$\begin{array}{lllllll}
f(x_0) = & f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(x_0 - x_k) & +\frac{f^{(2)}(x_k)}{2!}(x_0 - x_k)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(x_0 - x_k)^n \\
f(x_1) = & f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(x_1 - x_k) & +\frac{f^{(2)}(x_k)}{2!}(x_1 - x_k)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(x_1 - x_k)^n \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_{k-1}) = & f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(x_{k-1} - x_k) & +\frac{f^{(2)}(x_k)}{2!}(x_{k-1} - x_k)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(x_{k-1} - x_k)^n \\
f(x_{k+1}) = & f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(x_{k+1} - x_k) & +\frac{f^{(2)}(x_k)}{2!}(x_{k+1} - x_k)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(x_{k+1} - x_k)^n \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_n) = & f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(x_n - x_k) & +\frac{f^{(2)}(x_k)}{2!}(x_n - x_k)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(x_n - x_k)^n
\end{array} \tag{3.13}$$

Jak je vidět v soustavě (3.13) je n neznámých. Vyřešením soustavy zjistíme přibližnou hodnotu derivace v bodě x_k. Pokud použijeme ekvidistantní síť, kde vzdálenost mezi dvěma sousedními body je vždy stejná, pak lze psát h místo $x_i - x_{i-1}$. Následně soustavu přepíšeme do tvaru (3.14).

$$\begin{array}{lllllll}
f(x_0) & = f(x_k) & +\frac{f^{(1)}(x_k)}{1!}((0-k)h) & +\frac{f^{(2)}(x_k)}{2!}((0-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((0-k)h)^n \\
f(x_1) & = f(x_k) & +\frac{f^{(1)}(x_k)}{1!}((1-k)h) & +\frac{f^{(2)}(x_k)}{2!}((1-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((1-k)h)^n \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_{k-1}) & = f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(-1h) & +\frac{f^{(2)}(x_k)}{2!}(-1h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(-1h)^n \\
f(x_{k+1}) & = f(x_k) & +\frac{f^{(1)}(x_k)}{1!}(1h) & +\frac{f^{(2)}(x_k)}{2!}(1h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(1h)^n \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_n) & = f(x_k) & +\frac{f^{(1)}(x_k)}{1!}((n-k)h) & +\frac{f^{(2)}(x_k)}{2!}((n-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((n-k)h)^n
\end{array} \tag{3.14}$$

Přesunutím všech známých hodnot doleva dostaneme soustavu rovnic (3.15):

$$\begin{array}{llllll}
f(x_0) - f(x_k) = & \frac{f^{(1)}(x_k)}{1!}((0-k)h) & +\frac{f^{(2)}(x_k)}{2!}((0-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((0-k)h)^n & \\
f(x_1) - f(x_k) = & \frac{f^{(1)}(x_k)}{1!}((1-k)h) & +\frac{f^{(2)}(x_k)}{2!}((1-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((1-k)h)^n & \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_{k-1}) - f(x_k) = & \frac{f^{(1)}(x_k)}{1!}(-1h) & +\frac{f^{(2)}(x_k)}{2!}(-1h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(-1h)^n & \\
f(x_{k+1}) - f(x_k) = & \frac{f^{(1)}(x_k)}{1!}(1h) & +\frac{f^{(2)}(x_k)}{2!}(1h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}(1h)^n & \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
f(x_n) - f(x_k) = & \frac{f^{(1)}(x_k)}{1!}((n-k)h) & +\frac{f^{(2)}(x_k)}{2!}((n-k)h)^2 & +... & +\frac{f^{(n)}(x_2)}{n!}((n-k)h)^n &
\end{array} \tag{3.15}$$

Soustavu (3.15) lze přepsat jako násobení matice s vektorem $b = Ay$, kde v rovnici (3.16) b představuje vzdálenosti jednotlivých bodů od bodu x_k, A je matice koeficientů a y je matice neznámých derivací v bodě x_k :

$$b = \begin{bmatrix} f(x_0) - f(x_k) \\ f(x_1) - f(x_k) \\ \vdots \\ f(x_{k-1}) - f(x_k) \\ f(x_{k+1}) - f(x_k) \\ \vdots \\ f(x_n) - f(x_k) \end{bmatrix} A = \begin{bmatrix} \frac{((0-k)*h)^1}{1!} & \frac{((0-k)*h)^2}{2!} & \dots & \frac{((0-k)*h)^n}{n!} \\ \frac{((1-k)*h)^1}{1!} & \frac{((1-k)*h)^2}{2!} & \dots & \frac{((1-k)*h)^n}{n!} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{((-1)*h)^1}{1!} & \frac{((-1)*h)^2}{2!} & \dots & \frac{((-1)*h)^n}{n!} \\ \frac{((1)*h)^1}{1!} & \frac{((1)*h)^2}{2!} & \dots & \frac{((1)*h)^n}{n!} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{((n-k)*h)^1}{1!} & \frac{((n-k)*h)^2}{2!} & \dots & \frac{((n-k)*h)^n}{n!} \end{bmatrix} y = \begin{bmatrix} f^{(1)}(x_k) \\ f^{(2)}(x_k) \\ \vdots \\ f^{(k)}(x_k) \\ f^{(k+1)}(x_k) \\ \vdots \\ f^{(n)}(x_k) \end{bmatrix} \tag{3.16}$$

3.7.1 Inverzní matice

Vynásobením rovnice $b = Ay$ inverzní maticí vznikne rovnice $A^{-1}b = y$. Nevýhodou tohoto převodu je potřeba výpočtu inverzní matice. Jak známo výpočet inverzní matice je velice náročný, dokonce náročnější než jakýkoliv známý algoritmus řešení soustavy rovnic. Pro použitelnost této metody je nutné řešit inverzní matici, tak aby byla inverzní matice znovupoužitelná. Například parametrizací nebo převedením parametrů na levou stranu. Nevýhoda náročného výpočtu je pak odstraněna vícenásobným použitím jedné inverzní matice v programu. Výpočet inverzní matice lze provést pouze při startu programu nebo při překladu. Pro výpočet inverzní matice je možné použít analytické řešení nebo pseudoinverzní matici. Aby inverzní matice mohla být znovupoužitelná je potřeba volit ekvidistantní síť nebo síť s periodickým opakováním vzdáleností jednotlivých bodů. Vzorec (3.17) zobrazuje příklad posloupností bodů, kde se pravidelně vzdálenost mění od předcházejícího bodu s periodou dva. Například body $X = \{1, 2, 4, 5, 7, 8, 10\}$. Výpočet inverzní matice lze provést algoritmem Gaussovy eliminace nebo pomocí Moore-Penrose pseudoinverzní matice.

$$x_i - x_{i-1} = \begin{cases} 2 & \text{pro } i \text{ je liché} \\ 1 & \text{pro } i \text{ je sudé} \end{cases} \tag{3.17}$$

Numerické řešení: nevýhodou soustavy (3.16) je závislost výpočtu inverzní matice na vzdálenosti jednotlivých bodů. Závislost lze odstranit převedením parametru do vektoru y. Pro jednoduchost použijme ekvidistantní síť Nyní dostaneme soustavu (3.18) Soustava lze vyřešit numerickou metodou. Například Gaussovou eliminací nebo pseudoinverzní maticí.

$$b = \begin{bmatrix} f(x_0) - f(x_k) \\ f(x_1) - f(x_k) \\ \vdots \\ f(x_{k-1}) - f(x_k) \\ f(x_{k+1}) - f(x_k) \\ \vdots \\ f(x_n) - f(x_k) \end{bmatrix} A = \begin{bmatrix} (0-k)^1 & (0-k)^2 & \dots & (0-k)^n \\ (1-k)^1 & (1-k)^2 & \dots & (1-k)^n \\ \vdots & \vdots & \ddots & \vdots \\ (-1)^1 & (-1)^2 & \dots & (-1)^n \\ (1)^1 & (1)^2 & \dots & (1)^n \\ \vdots & \vdots & \ddots & \vdots \\ (n-k)^1 & (n-k)^2 & \dots & (n-k)^n \end{bmatrix} y = \begin{bmatrix} f^{(1)}(x_k) * \frac{h^1}{1!} \\ f^{(2)}(x_k) * \frac{h^2}{2!} \\ \vdots \\ f^{(k)}(x_k) * \frac{h^k}{k!} \\ f^{(k+1)}(x_k) * \frac{h^{k+1}}{(k+1)!} \\ \vdots \\ f^{(n)}(x_k) * \frac{h^n}{n!} \end{bmatrix} \tag{3.18}$$

Po dosazení hodnot za b a vynásobení $A^{-1} * b = y$ získáme vektor y obsahující derivace závislé na vektoru c. Pokud nyní vytvoříme vektorový součin vektoru y s vektorem v získáme

požadované derivace $f^{(i)}(x_k)$. Vektor v je inverzním zobrazením vektoru c, vztah mezi nimi lze vyjádřit jako $v = c^{-1}$ Vektor v je jednoduše vyčíslitelný při běhu programu.

$$c = \begin{bmatrix} \frac{h^1}{1!} \\ \frac{h^2}{2!} \\ \vdots \\ \frac{h^k}{k!} \\ \frac{h^{k+1}}{(k+1)!} \\ \vdots \\ \frac{h^n}{n!} \end{bmatrix} \quad v = \begin{bmatrix} \frac{1!}{h^1} \\ \frac{2!}{h^2} \\ \vdots \\ \frac{k!}{h^k} \\ \frac{(k+1)!}{h^{k+1}} \\ \vdots \\ \frac{n!}{h^n} \end{bmatrix}$$

Analytické řešení: Pomocí analytického řešení inverzní matice A (3.16) je možné vyřešit soustavu tak, aby vznikly rovnice o několika neznámých. Do těchto rovnic je potom možné dosazovat. Metoda je nevýhodná při velkém počtu derivací, kdy je potřeba analyticky velký počet rovnic. Výsledky metody jsou docela přesné, ale použitelnost klesá vysokým počtem matic, pro případný výpočet dopřednou, kombinovanou, zpětnou metodou. Uvažujme rozmístění bodů na ekvidistantní síti. Soustavu vyřešme obecně s parametrem $h = x_i - x_{i-1}$ určujícím vzdálenost bodů na ekvidistantní síti. Pro analytický výpočet inverzní matice lze použít Cramerovo pravidlo (3.19), kde $f^{(n)}$ je n-tá derivace v daném bodě.

$$f^{(n)} = \frac{|A_n|}{|A|} \tag{3.19}$$

Příklad (3.20) ukazuje postup analytického výpočtu na ekvidistantní síti pěti bodů x_0, x_1, x_2, x_3, x_4. Bodem ve kterém chceme znát derivace je bod $x_2 = x_k$. Použitím pouze pěti bodů lze získat maximálně čtyři derivace funkce v daném bodě. Pro příklad jsou požity hodnoty $n = 4$, $k = 2$ a matice A_2 pro výpočet druhé derivace $f^{(2)}$.

$$A = \begin{bmatrix} \frac{((0-k)*h)^1}{1!} & \frac{((0-k)*h)^2}{2!} & \frac{((0-k)*h)^3}{3!} & \frac{((0-k)*h)^n}{n!} \\ \frac{((-1)*h)^1}{1!} & \frac{((-1)*h)^2}{2!} & \frac{((-1)*h)^3}{3!} & \frac{((-1)*h)^n}{n!} \\ \frac{((1)*h)^1}{1!} & \frac{((1)*h)^2}{2!} & \frac{((1)*h)^3}{3!} & \frac{((1)*h)^n}{n!} \\ \frac{((n-k)*h)^1}{1!} & \frac{((n-k)*h)^2}{2!} & \frac{((n-k)*h)^3}{3!} & \frac{((n-k)*h)^n}{n!} \end{bmatrix} = \begin{bmatrix} \frac{(-2h)^1}{1!} & \frac{(-2)h^2}{2!} & \frac{(-2h)^3}{3!} & \frac{(-2h)^n}{n!} \\ \frac{(-h)^1}{1!} & \frac{(-h)^2}{2!} & \frac{(-h)^3}{3!} & \frac{(-h)^n}{n!} \\ \frac{(h)^1}{1!} & \frac{(h)^2}{2!} & \frac{(h)^3}{3!} & \frac{(h)^n}{n!} \\ \frac{(2h)^1}{1!} & \frac{(2h)^2}{2!} & \frac{(2h)^3}{3!} & \frac{(2h)^n}{n!} \end{bmatrix} \tag{3.20}$$

$$A_2 = \begin{bmatrix} \frac{((0-k)*h)^1}{1!} & a & \frac{((0-k)*h)^3}{3!} & \frac{((0-k)*h)^n}{n!} \\ \frac{((-1)*h)^1}{1!} & b & \frac{((-1)*h)^3}{3!} & \frac{((-1)*h)^n}{n!} \\ \frac{((1)*h)^1}{1!} & c & \frac{((1)*h)^3}{3!} & \frac{((1)*h)^n}{n!} \\ \frac{((n-k)*h)^1}{1!} & d & \frac{((n-k)*h)^3}{3!} & \frac{((n-k)*h)^n}{n!} \end{bmatrix} \tag{3.21}$$

Po využití vzorce (3.19) vznikne rovnice (3.22) s parametry a, b, c, d, h. Parametr h je vzdálenost dvou nejbližších bodů. Výpočet ostatních parametrů je zobrazen ve vzorci (3.22)

$$f^{(2)}(a, b, c, d, h) = \frac{\frac{-1}{12}(a + d) + \frac{4}{3}(b + c)}{h^2} \tag{3.22}$$

Kde:

$$\begin{aligned} a &= f(x_0) - f(x_2) \\ b &= f(x_1) - f(x_2) \\ c &= f(x_3) - f(x_2) \\ d &= f(x_4) - f(x_2) \end{aligned}$$

3.7.2 Řešením soustavy rovnic

Další možností je řešit soustavu lineárních rovnic za běhu programu. Při každém kroku simulace či integrace je sestavení a vyřešení soustavy rovnic nákladnější než **opakované** použití jednou vypočtené inverzní matice. Pokud není garantována žádná perioda bodů pro prováděný výpočet, pak nám bohužel nezbývá než řešit soustavu rovnic o n neznámých v každém kroku. Metoda je nákladná na procesorový čas, proto je dobré se této metodě vyhnout. Řešení soustavy rovnic lze řešit například Gauss-Saidelovou metodou nebo Newtonovou metodou.

Kapitola 4

Paralelní výpočet a jeho využití v metodě Taylorovy řady

Paralelní systém je definován jako množina aritmeticko-logických jednotek (ALU) pracujících na stejném úkolu (Problému). Množina ALU jednotek může být více počítačů, více procesorů, více jader, či jejich kombinace. Problém je rozdělen na několik menších částí, a pak je vykonáván na různých ALU jednotkách. Úsilím paralelizace je především snížení celkové doby výpočtu současným zpracováním většího množství dat. ALU jednotky mezi sebou komunikují pomocí sdílené paměti, nebo pomocí zasílaní zpráv po sběrnici.

U klasických jednoprocesorových algoritmů se udává složitost algoritmus, která určuje jak je algoritmus časově efektivní. U paralelních algoritmů se udávají tři údaje časová složitost, prostorová složitost, a poslední složitost je roznásobením časové a prostorové složitosti. Časová složitost je zde čas spotřebovaný pro vyřešení problému v závislosti na jeho velikosti. Prostorová složitost udává, kolik procesorů je potřeba pro vyřešení problému v závislosti na velikosti problému. Většinou velikost problému se udává jako n, tedy počet prvků nad kterými je úloha prováděna. Dalším důležitým parametrem paralelních algoritmů je škálovatelnost. Škálovatelnost, určuje o kolik více zdrojů je potřeba, a o kolik se sníží čas potřebný pro výpočet, při přidání jednoho procesoru do výpočtu. Škálovatelnost je převážně dána synchronizací mezi jednotlivými ALU jednotkami. Častým důvodem je nerovnoměrná zátěž jednotlivých ALU jednotek.

4.1 Úrovně paralelizmu

Paralelizmus lze rozdělit do několika úrovní.

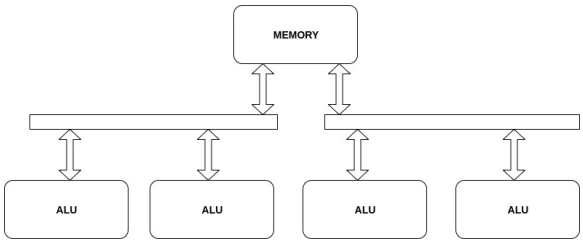
- uvnitř instrukcí – nejjemnější typ paralelismu nacházející se na úrovni bitů, příkladem použití jsou 8, 16, 32, 64 bitové procesory
- mezi instrukcemi – Procesor je velice složitá jednotka, obsahuje jednotky pro sčítání, odčítání, násobení, dělení, a jiné. Většina operací používá pouze jednu tuto jednotku. Proto se zavedlo souběžné provádění některých instrukcí, čímž se značně zvýšil podíl pracujících podkomponent v procesoru. Tento druh paralelizmu není programátorovi viditelný.
- mezi vlákny – procesy paralelního systému se rozdělují na výpočetní vlákna. Vlákna pracují se sdílenou pamětí. Vlákna mohou běžet na všech ALU jednotkách v systému.

- mezi procesy – Problém je řešen na několika procesorech současně. Každý z procesů disponuje vlastní pamětí, kterou nesdílí s jinými procesy.

Paralelizmus na úrovni instrukcí a mezi instrukcemi není těžké začít používat. Programátor nepotřebuje žádné větší znalosti o dané architektuře paralelního systému, stačí k tomu pouze podpora překladače. Překladač seřadí instrukce, tak aby jednotlivé instrukce byly co nejméně závislé na předcházejících instrukcích. Využití paralelizmu na úrovni vláken, nebo procesů je třeba zohlednit paralelní konstrukci, přitom je nutné také počítat s nutnou komunikací mezi jednotlivými bloky.

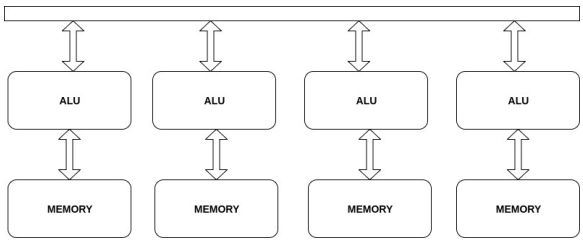
4.2 Komunikace paralelních systémů

V systémech se **sdílenou pamětí** sdílejí všechny výpočetní jednotky stejný adresový prostor. Jednotlivé ALU komunikují pomocí jedné sdílené paměti. Komunikace probíhá čtením a zápisem dat do této paměti. U tohoto přístupu je nutná synchronizace čtení/zápisu do sdílené paměti. U tohoto přístupu je pomyslné úzké hrdlo láhve komunikace s pamětí.



Obrázek 4.1: sdilená paměť

V systémech s **distribuovanou pamětí** komunikace probíhá přeposíláním zpráv jednotlivým procesorům. V těchto systémech má každá ALU jednotka svoji paměť ve které jsou uložena potřebná data. U tohoto přístupu je pomyslné úzké hrdlo láhve komunikace mezi jednotlivými ALU jednotkami. Tento problém má dvě roviny. Prvním případem je nízký počet propojení. Zde dochází k problému s malou přenosovou propustností sítě mezi ALU jednotkami. Druhým extrémem je příliš velký počet propojení. Zde je sice velká propustnost sítě mezi ALU jednotkami, ale jsou zde velké nároky na prostor na čipu.



Obrázek 4.2: sdilená paměť

4.3 Flynnova klasifikace

Víceprocesorové počítačové systémy se podle Flynnovy klasifikace dělí do několika kategorií. Častá klasifikace těchto systémů je dělení podle uspořádání operační paměti. [13]

- SISD (Single Instruction Single Data) – Nejedná se o paralelní systém, ale o jednoprocesorový systém. Jeden Procesor provádí sekvenční posloupnost instrukcí nad daty uloženými v paměti.
- SIMD (Single Instruction Multiple Data) – Několik procesorů se společným řízením pracuje paralelně nad různými daty. Komunikace probíhá pomocí zasílání zpráv nebo přes sdílenou paměť. Díky existenci jedné řídící jednotky jsou procesory implicitně sesynchronizovány. Není tak univerzální jako následující varianty.
- MISD (Multiple Instruction Single Data) – Stejná data se zpracovávají v několika jednotkách. Tento typ architektur je využíván jen velmi zřídka.
- MIMD (Multiple Instruction Multiple Data) – Paralelní nezávisle řízené procesory, které pracují nad různými daty. Procesory mezi sebou mohou komunikovat pomocí zasílání zpráv nebo přes sdílenou paměť. Univerzálnost této architektury je největší ze zmíněných. Většina dnešních moderních paralelních architektur patří právě do této třídy.

4.4 Využití Paralelizace v metodě Taylorovy řady

Navrhnout rozsáhlý systém je velice obtížné. Musí zde být efektivně propojeno několik výpočetních uzlů a efektivně vytvořeno řízení těchto uzlů. V našem případě je ALU jednotka velice specifická jednotka počítající integrál pomocí Taylorova řady. U metody Taylorovy řady, kdy aproximujeme integrál pomocí uzlových bodů z derivační funkce, existuje několik možností paralelizace.

Na úrovní procesů vláken existují dvě myslitelné možnosti paralelizace. První možností je využití definice Riemanova integrálu. Principem je rozdělit nejvyšší integrál na několik částí. Každou z těchto částí vypočítat a následně vše sečíst dohromady. Druhou možností je paralelní výpočet uzlových bodů. Tyto uzlové body mohou být také integrály. Uzlové body, ze kterých je aproximován integrál, jsou na sobě nezávislé a výpočet jednoho uzlového bodu neovlivňuje výpočet druhého uzlového bodu. Jedná se vlastně o výpočet hodnot $f(x)|x \in \{u_1, u_2, ..., u10\}$ zobrazených v obrázcích 3.7, 3.8, a 3.9.

4.4.1 Paralelní výpočet integrálu z uzlových bodů

Další možností paralelizace je vektorové, tedy SIMD (single instruction multiple data). Metoda Taylorovy řady pro výpočet integrálu z aproximačních bodů probíhá v několika krocích. Výpočet derivací vyšších řádů je uveden v kapitole 3.7 Pro ukázku zvolme aproximaci čtyř derivací v bodě x_2 pomocí pěti bodů x_0, x_1, x_2, x_3, x_4 mající vlastnost $x_i - x_{i-1} = h$, kde h je konstantní krok. Vznikne tedy soustava

$$\begin{aligned} a &= f(x_0) - f(x_2) \\ b &= f(x_1) - f(x_2) \\ c &= f(x_3) - f(x_2) \\ d &= f(x_4) - f(x_2) \end{aligned} \tag{4.1}$$

$$\begin{aligned} d_1 &= f^{(1)}(x_2) * \frac{h^1}{1!} = (G_{11} * a + G_{12} * b + G_{13} * c + G_{14} * d) \\ d_2 &= f^{(2)}(x_2) * \frac{h^2}{2!} = (G_{21} * a + G_{22} * b + G_{23} * c + G_{24} * d) \\ d_3 &= f^{(3)}(x_2) * \frac{h^3}{3!} = (G_{31} * a + G_{32} * b + G_{33} * c + G_{34} * d) \\ d_4 &= f^{(4)}(x_2) * \frac{h^4}{4!} = (G_{41} * a + G_{42} * b + G_{43} * c + G_{44} * d) \end{aligned} \tag{4.2}$$

Hodnota G_{ij} ve vzorci (4.2) reprezentuje hodnotu v inverzní matici G uloženou na pozici i, j. Jelikož při výpočtu integrálu není potřeba vypočítat derivace, nýbrž jen hodnoty, které se nachází v levé části vzorce (4.2). Je zlomek uveden v levé části vzorce. Jedná se vlastně o koeficienty původní Taylorovy řady. V následující tabulce 4.1 každý sloupec představuje jednu ALU jednotku. Jednotky mohou být řízeny jak synchroně, tak asynchroně. každý řádek reprezentuje aktuálně prováděnou operaci v procesoru. Tabulka představuje pouze pravou část vzorců (4.1) (4.2). Algoritmus má celkem 8 kroků. Na konci algoritmu jsou výsledky uloženy v proměných t_1, t_2, t_3, t_4.

číslo Operace	**Jednotka 1**	**Jednotka 2**	**Jednotka 3**	**Jednotka 4**
1	$a = f(x_0) - f(x_2)$	$b = f(x_1) - f(x_2)$	$c = f(x_3) - f(x_2)$	$d = f(x_4) - f(x_2)$
2	$t_{11} = G_{11} * a$	$t_{21} = G_{21} * a$	$t_{31} = G_{31} * a$	$t_{41} = G_{41} * a$
3	$t_{12} = G_{12} * b$	$t_{22} = G_{22} * b$	$t_{32} = G_{32} * b$	$t_{42} = G_{42} * b$
4	$t_{13} = G_{13} * c$	$t_{23} = G_{23} * c$	$t_{33} = G_{33} * c$	$t_{43} = G_{43} * c$
5	$t_{14} = G_{14} * d$	$t_{24} = G_{24} * d$	$t_{34} = G_{34} * d$	$t_{44} = G_{44} * d$
6	$d_1 = t_{11} + t_{12}$	$d_2 = t_{21} + t_{22}$	$d_3 = t_{31} + t_{32}$	$d_4 = t_{41} + t_{42}$
7	$d_1 = d_1 + t_{13}$	$d_2 = d_2 + t_{23}$	$d_3 = d_3 + t_{33}$	$d_4 = d_4 + t_{43}$
8	$d_1 = d_1 + t_{14}$	$d_2 = d_2 + t_{24}$	$d_3 = d_3 + t_{34}$	$d_4 = d_4 + t_{44}$

Tabulka 4.1: Paralelní výpočet koeficientů Taylorovy řady

Nejefektivnější výpočet integrálu Taylorovým polynomem z je sekvenční, algoritmus zobrazen v tabulce 4.2 nelze již provádět asynchroně. Jelikož v systému jsou již paralelní jednotky. Je vhodné jejich použití, a využití stromového sčítaní. Ve vzorci (4.3) a tabulce 4.2 je zobrazen výpočet integrálu funkce v bodě $F(x_2 + j)$, kde $F(x_2)$ je hodnota integrálu v bodu $F(x_2)$, pokud je to počátek aproximace, pak hodnota $F(x_2)$ je rovna 0. Hodnota j představuje velikost integračního kroku. Pro výpočet je potřeba ještě spočítat hodnoty $k_i = \frac{j^i}{h^{i-1}}$, tento výpočet je možné provádět paralelně s výpočtem zobrazeným v tabulce 4.1.

$$F(x_2 + j) = F(x_2) + f(x_2) * \frac{j}{1!} + d_1 * \frac{j^2}{2 * h} + d_2 * \frac{j^3}{3 * h^2} + d_3 * \frac{j^4}{4 * h^3} + d_4 * \frac{j^5}{5 * h^4} \tag{4.3}$$

číslo Operace	**Jednotka 1**	**Jednotka 2**	**Jednotka 3**	**Jednotka 4**
1	$t_1 = d_1/2$	$t_2 = d_2/3$	$t_3 = d_3/4$	$t_4 = d_4/5$
2	$t_1 = t_1 * k_1$	$t_2 = t_2 * k_2$	$t_3 = t_3 * k_3$	$t_4 = t_4 * k_4$
3	$t_1 = t_1 + t_2$		$t_3 = t_3 + t_4$	
4	$t_1 = t_1 + t_3$			
5	$u = f(x_2) * k_0$			
6	$u = u + t_1$			
7	$F(x_2) + u$			

Tabulka 4.2: Paralelních integrálů pomocí Taylorovy řady

Výpočet integrálu je paralelní jen do kroku 3, dále pak je pouze sekvenční. V kroku 7 je již vypočítán integrál v bodě $f(x_2 + j)$

Kapitola 5

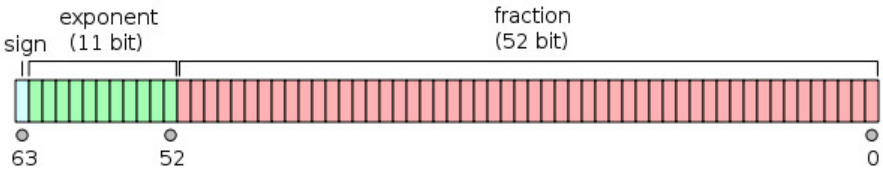
Numerické výpočty

5.1 Datové typy

V podstatě existují dva základní datové typy s desetinnou čárkou. Prvním typem je datový typ s pevnou desetinnou čárkou. Je to jednoduchý typ, kde se určí napevno pozice desetinné čárky. Tento datový typ má jednu velkou nevýhodu: není možné měnit dynamicky rozsah podle velikosti ukládaných čísel.

Obrázek 5.1: číslo s pevnou desetinnou čárkou

Druhým typem je datový typ s plovoucí desetinnou čárkou. Číslo s plovoucí desetinnou čárkou je rozděleno na čtyři části. První částí je jednobitová informace o znaménku. Druhou částí je exponent. Třetí částí je jednobitová informace o normalizaci čísla. Pokud je zde nula, číslo není normalizováno. Je tedy příliš malé nebo příliš velké aby se vešlo do datového typu. Jeho exponent se nevejde do části pro exponent. Poslední částí je setinná část čísla. Číslo má tedy tvaru $(\pm 1.101e111)_2$.



Obrázek 5.2: double podle standardu IEEE (wikipedia)

Klasické datové typy s plovoucí řádovou čárkou mají příliš malou desetinnou část i exponenciální část. Proto jsou pro složitější integrování nepoužitelné. Maximální přesnost má datový typ *long double*, který má pouze 63 binárních desetinných míst, což odpovídá zhruba 18.96 desetinným místům. Je tedy nutné zvýšit počet desetinných míst. V implementaci jsem použil k zvýšení přesnosti knihovnu **gmp**[17]. Knihovna umožňuje na začátku programu zvolit velikost setinné části. Část pro uložení exponentu je větší než u klasických datových typů s plovoucí řádovou čárkou. Součástí knihovny nejsou již implementace funkcí jako je $sin(x), cos(x), e^x$, pro $x \in R$, tyto funkce musely být doimplementovány.

Zajímavým datovým typem je datový typ signed bit popsaný publikací [16]. Jedná se vlastně o Boothovo překódování s radixem 2. Tento typ přidává před každý bit bit se znaménkem. Vyšší paměťově nároky jsou kompenzovány absencí přenosu při sčítání do vyšších bitů. Při zvětšování typu nedochází k prodlužování kritických cest v designu hardwarové komponenty. Boothovo překódování s radixem 2 je náročné na zdroj, jelikož místo jednoho bitu je potřeba udržovat dva bity. Proto bych spíše doporučil Boothovo překódování s vyšším radixem. Například 8 či 32. Klasické datové typy

$$(1010)_2 = (10)_{10}$$

Datový typ Signed bit

$$(01\ 01\ 11\ 00)_2 = (01\ 00\ 01\ 00)_2 = (10)_{10}$$

5.2 Chyby v numerických výpočtech

V numerických výpočtech vznikají chyby aproximace funkce, nebo numerické chyby. Všechny druhy chyb jsou nežádoucí a mohou výsledek zdlouhavého výpočtu degradovat na nepoužitelné číslo. Proto je nutné zabývat se jednotlivými druhy chyb a snažit se odstranit jejich příčiny.

Existuje mnoho dalších chyb, ke kterým může dojít. Jednou z těchto chyb je chyba formulace matematické úlohy [2]. Při zkoumání jevů jsme nuceni úlohu zjednodušit. Není v silách jak lidských tak výpočetních vyřešit nezjednodušenou úlohu. Další chyby jsou způsobeny tím, že některé hodnoty lze určit pouze přibližně. Měřící přístroje mají odchylky a vstupní hodnoty simulace či integrálu není tedy určena přesně, ale s nějakou nepřesnosti. Další chyby souvisí z číselnou soustavou ve které číslo zapisujeme. Tyto hodnoty musíme zaokrouhlit nebo oříznout. Hodnoty jako například π mají neukončený desetinný rozvoj, což způsobuje chybu. Nebo číslo $0,1$ v desítkové soustavě má ukončený rozvoj, ale zapíšeme li ho v dvojkovém doplňkovém kódu jeho rozvoj bude neukončený.

Existují dvě metriky pro určení chyby: absolutní a relativní. První metrikou je metrika absolutní chyby. Definujme množinu A jako okolí bodu a a bod $r \in A$, který se liší dostatečně málo od bodu a. Bod r budeme nahrazovat bodem a ve výpočtech. Absolutní chyba se vypočítá jako absolutní hodnota z rozdílu čísel a a r $\Delta = |a - r|$. Bod a často neznáme, proto nelze absolutní chybu určit. Lze ji však odhadnout. Mějme například číslo π víme, že $3.14 < \pi < 3.15$ lze tedy odhadnout absolutní chybu jako $\Delta = 0.01 = 3.15 - 3.14$. Pak aproximujme bod π bodem 3.14. Definujme množinu $B =< 3.14 - \Delta; 3.14 + \Delta >$. Je patrné, že $3.14 \in A \subset B$. A tudíž $\pi \in B$

Další používanou metrikou je relativní chyba δ. Relativní chyba, zobrazená v rovnici (5.1), je poměr absolutní chyby Δ k hodnotě čísla a. Relativní chyba tedy reprezentuje procentuální odchylku od skutečné hodnoty.

$$\delta = \frac{|\Delta|}{|a|} \tag{5.1}$$

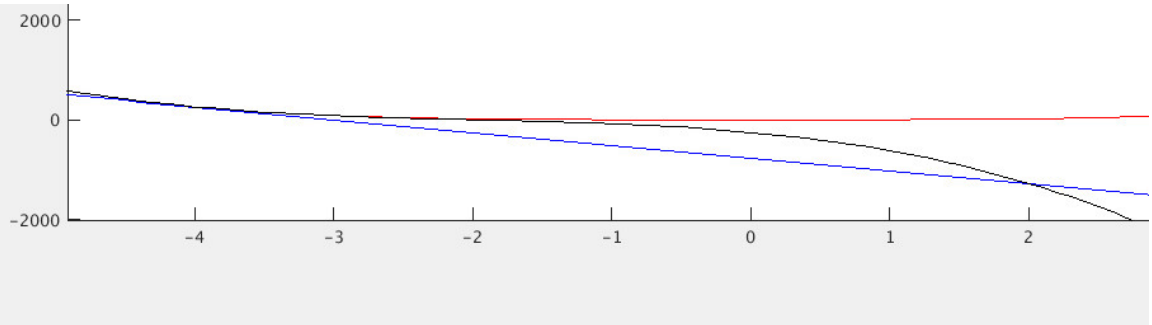
5.2.1 Chyba aproximace

V aproximačních vzorcích funkcí se často objevují nekonečné řady. Obecně nelze nekonečný proces popsat konečným počtem kroků. Musí dojít k oříznutí výsledku u nějakého členu n. Tím je způsobena chyba aproximace. Chyba aproximace $E(x)$ se projevuje jako rozdíl původní funkce $f(x)$ a aproximační funkce $A(x)$. Důležitou hodnotou je maximální absolutní hodnota funkce chyby $E(x)$ na dané množině J, pro kterou aproximace má ještě smysl. Matematicky ji lze vyjádřit (5.2).

$$\begin{aligned} E(x) &= f(x) - A(x) \\ ERR &= max\{E(x)|E'_x(x) = 0\} \end{aligned} \tag{5.2}$$

Při numerických výpočtech není možné většinou chybu aproximace získat, proto se používá odhad aproximační chyby. Pro odhad aproximační chyby $E(x)$ se používá metoda porovnáni dvou aproximačních metod. Může jít o jinou aproximační metodu, nebo pouze o zjemnění kroku v metodě. V případě metody Taylorovy řady lze jako druhou metodu použít Taylorovu řadu s vyšším počtem použitých derivací při výpočtu. Pro odhad chyby je použit vzorec pro výpočet přesné chyby. Pouze místo původní funkce $f(x)$ se použije výsledek přesnější aproximační metody. Jelikož se jedná pouze o odhad chyby, může být tento odhad velice nepřesný. Příkladem buď: Funkce $f(x)$ a její aproximace pomocí Taylorova polynomu prvního řádu $a1(x)$ a pomocí Taylorova polynomu třetího řádu $a3(x)$. Oba Taylorovy polynomy jsou aproximovány z bodu -4. Při použití nastíněné metody k odhadu chyby může dojít k velmi nepřesnému odhadu chyby. Tento problém je zobrazen na grafu 5.3, kde je odhad v bodě 2 velmi nepřesný. V tomto případě budeme v numerickém výpočtu předpokládat malou chybu, ale skutečná chyba bude enormní. Detail funkcí je zobrazen ve vzorci (5.2.1). Dále na obrázku 5.3 je vidět relativně přesnou aproximaci do bodu $-3,5$ pomocí funkce $a1(x)$ a do bodu -1 pomocí funkce $a3(x)$. Na obrázku5.3 jsou zobrazeny funkce následovně $f(x)$ červená, $a1(x)$ modrá a $a3(x)$ je černá.

$$\begin{aligned} f(x) &= x^4 \\ a1(x) &= 256 - \frac{256}{1!} * (x-4) \\ a3(x) &= 256 - \frac{256}{1!} * (x-4) + \frac{192}{2!} * (x-4)^2 - \frac{96}{3!} * (x+4)^3 \end{aligned} \tag{5.3}$$



Obrázek 5.3: chyba aproximace

5.2.2 Numerické chyby

Některá čísla mají větší počet platných číslic než čísla, se kterými jsme schopni počítat. Proto jsou tato čísla oříznuta nebo zaokrouhlena na čísla s počtem číslic, se kterým jsme schopni již počítat. Pří provádění operací s čísly se přenáší chyba vstupních čísel na výstupní čísla. Pro některé operace jsou známy techniky pro zmenšení chyby. U sčítání je vhodné sčítat od nejmenšího sčítance po největší sčítanec.

Definujme okolí bodu a jako množinu A a okolí bodu b jako množinu B. Pak pro $k = a \circ b$ platí $K = (A \circ B)$ a $k \in K$. U většiny aplikací nemusíme brát v úvahu celé množiny, stačí pouze brát v úvahu krajní body. Z krajních bodů množin A, B je jednoduché spočítat krajní body množiny K. Například u operace sčítání lze množinu $K =< k_1; k_2 >$ určit výběrem maxima a minima ze součtu krajních hodnot množin.

$$Z = \{min(A) + min(B); min(A) + max(B); max(A) + min(B); max(A) + max(B)\}$$
$$k_1 = min(Z)$$
$$k_2 = max(Z)$$

Existují však operátory u kterých toto zjednodušení není možné. Například absolutní hodnota. Zde dojde k převrácení záporných mezních hodnot do kladných hodnot. Například mějme číslo $k \in < -1; 1 >$. Pokud bychom chtěli výpočet chyby zjednodušit, dopustili bychom se chyby. Výsledek takového počínaní by byl $k \in < 1; 1 >$ což je chybné. Správný výsledek ke kterému bychom se měli dostat je $k \in < 0; 1 >$.

Kapitola 6

Implementace v SW

Pro implementaci v Softwaru jsme zvolil programovací jazyk C. Vzhledem k povaze výpočtů je potřeba se neomezovat jen na klasickou aritmetiku, ale použít víceslovní aritmetiku. Aby byla práce s aritmetikou co nejjednodušší vytvořil jsem modul "*variable.h*". Pomocí tohoto modulu je jednoduché přepínat mezi klasickou aritmetikou a víceslovní aritmetikou. Víceslovní aritmetika je implementována pomocí knihovny "*gmp*".

6.1 Návrh aplikace

Návrh programu je objektový. Program je škálován do několika modulů tak, aby jeho oprava případně přeprogramovaní jednotlivých částí bylo co nejlehčí.

V souboru **macros.h** jsou uloženy konstanty pro nastavení překladu projektu. Jednou z nejdůležitějších konstant jsou konstanty k určení typu a velikosti datového typu. Další důležitou konstantou je konstanta určující velikost inverzní matice. Velikost inverzní matice určuje kolik je potřeba prvních derivací pro aproximaci derivací vyšších řádů.

```
//src/header/macros.h
#ifndef _MACROS_H_
#define _MACROS_H_

//velikost inverzni matice (pocet derivaci)
#define INV_MATRIX_SIZE  4

//velikost typu
//0 - float, 1-double, 2-long double, 3 - dlouhy datovy typ (gmp),
#define TYPE_TYPE 3
// pocet bitu v datovem typu gmp (platny pouze, kdyz TYPE_TYPE = 3)
#define TYPE_SIZE 8192

#endif
```

Modul **variable.h** obsahuje implementaci sčítaní, odčítání, násobení, a dělení. Výběr implementace je proveden nastavením makra v modulu macros.h. Výhodou tohoto uspořádání je jednodušší ladění. Stačí přepnout makro a lze pracovat se standardními typy. Složitější funkce , jako jsou $sin(x)$, $cos(x)$, a e^x, jsou implementovány v extra modulu **int_math**. Zde jsou také uloženy konstanty pi a e potřebné pro výpočty. Modul je nutné

při startu programu inicializovat, kvůli převodu konstant z textové podoby do aktuálně zvoleného datového typu.

Při použití nestandardního datového typu je potřeba provádět inicializaci, alokaci, dealokaci proměnné s každým spuštěním funkce. Modul **program_stack** zabraňuje častým alokací a dealokacím. Modul alokuje souvislou paměť, kterou po vyžádání přiřazuje jednotlivým funkcím. Jde o obdobu klasického zásobníku v PC. Na začátku práce s modulem je nutné paměť inicializovat funkcí *program_stack_init*. Na konci práce je nutné paměť dealokovat funkcí *program_stack_destroy*. Níže je zobrazen ukázkový kód pro vytvoření dvou lokálních proměnných a, a b.

```
void fce(program_stack * stack){

const unsigned int prom_num = 2;
int_variable * tmp = program_stack_get(stack, prom_num);
int_variable * a = tmp + 0;
int_variable * b = tmp + 1;

program_stack_release(stack, prom_num);
}
```

Modul **matrix** se stará o výpočet inverzních matic. Inverzní matice slouží k aproximací derivací Taylorovým polynomem. Výpočet matic probíhá na začátku programu metodou Gaussovy eliminace. Start programu je proto docela pomalý. Dal by se urychlit uložením inverzní matice předem do souboru. Při startu programu by se jen inverzní matice načetla ze souboru.

Struktura vstupu se nachází v modulu **problem**. Modul obsahuje datovou strukturu představující integrál, který má program za úkol vypočítat. Samotný problém je popsán jako pole instrukcí. Seznam a význam jednotlivých instrukcí se nachází v modulu *equation*. Každá instrukce má maximálně tři operandy. Všechny operandy nemusí být vždy využity. V následujícím úryvku kódu je zobrazen datový typ *t_problem* představující řešený integrál. V tomto typu existuje několik proměnných. První proměnnou je *fce* jedná se o ukazatele na instrukce přestavující nejnižší řešenou funkci $f(\vec{x})$. Druhou proměnnou je pole ukazatelů na instrukce *int_start*, která představuje program pro jednotlivé úrovně integrálů k určení horní a dolní meze aktuálně počítaného integrálu. Proměnná *constants* obsahuje potřebné konstanty. Nejčastěji může jít o meze integrálů. Proměnná x je pole obsahující aktuální hodnoty x jednotlivých úrovní integrálů. Při inicializaci nižších integrálů se zde uloží dolní mez. Další proměnná *x_end* je pole přestavující horní mez jednotlivých úrovní integrálů. Při inicializaci nižšího integrálu se zde uloží horní mez integrálu. Předposlední proměnnou je *step_diff*. Zde je uložena hodnota kroku představujícího vzdálenost mezi derivacemi v jednotlivých úrovních integrálů. Poslední proměnou je *step_int* představující integrační krok v jednotlivých úrovních integrálů.

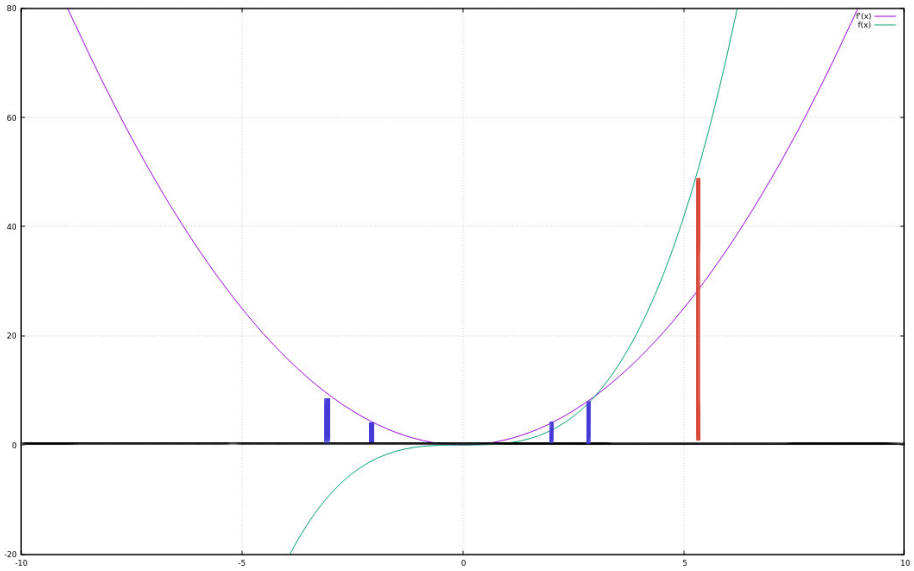
```
typedef struct{

//equation
int_equation_prog *  fce; //function
//setup variables (start and end of integral can be function)
int_equation_prog ** int_start;

int_variable * constants;
int_variable * x;
int_variable * x_end;
int_variable * step_diff;
int_variable * step_int;
unsigned int  dim; //dimensions
}t_problem;
```

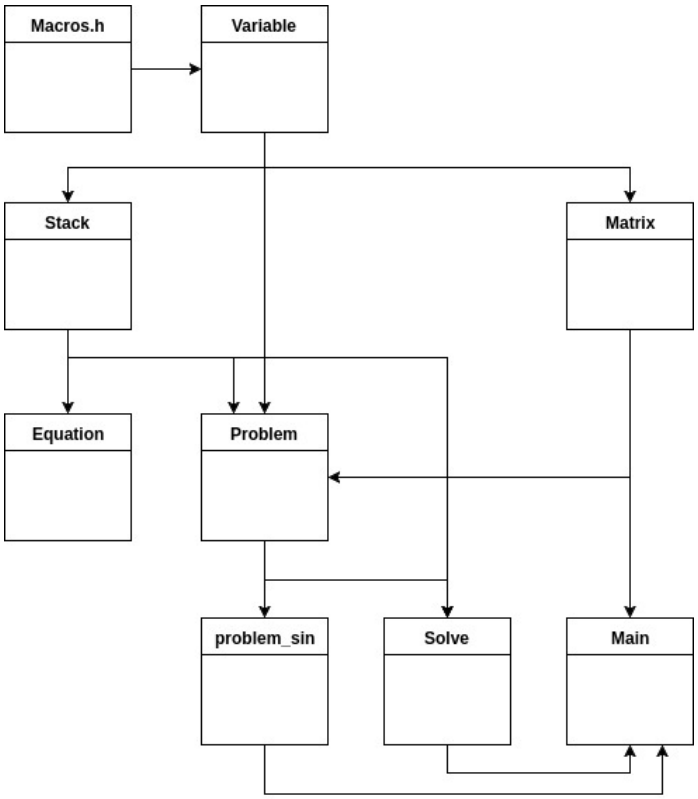
Nejdůležitějším modulem je modul **solve**. Zde se provádí samotný výpočet integrálu v několika krocích. Jedná se o rekurzivní vykonávání kódu. Na obrázku 6.1 je zobrazen derivační krok modrou barvou na funkci $f'(x)$ a integrační krok červenou barvou na funkci $f(x)$. Derivační krok slouží pro výpočet hodnot funkce a aproximaci hodnot derivací vyšších řádů. Integrační krok slouží pro výpočet integrálu funkce v bodě $f(x+h)$.

1. Pokud jsi na úrovni integrované funkce vykonej funkci a skonči
2. Nastav limity integrálu, tedy počátek a konec integrace.
3. S derivačním krokem vypočti n derivací . (zavolej funkci solve s parametrem dim-1)
4. Pomocí inverzní matice vypočítej derivace závislé na koeficientech.
5. Krok numerické integrace s integračním krokem.
6. Pokud není konec numerické integrace vrať se na bod 3 jinak skonči.



Obrázek 6.1: derivační a integrační krok

Na obrázku 6.2 je zobrazena závislost jednotlivých modulů. Modul *problem_sim* představuje již specifický problém, a lze nahradit překladovým modulem. Překladový modul by měl mít na práci překlad problému z textové podoby do reprezentace problému pomocí datového typu *t_probelm*.



Obrázek 6.2: Softwarový návrh modulů

6.2 Implementace sin, cos, exp

Pro implementaci těchto algoritmů jsem převážně využil literaturu [2]. Algoritmus je definován jako konečná posloupnost předepsaných úkonů. Pro výpočet všech zmíněných funkcí je potřeba nekonečných řad, a proto je nutné někde definovat ukončení výpočtu. Výpočet je ukončen, až když přičtením následujícího členu se výsledná hodnota nezmění. Ze všech vzorců (6.3) (6.6) (6.7) vyplývá, že každý další člen je v absolutní hodnotě menší než ten předchozí.

U funkcí sin a cos se veškerý výpočet převede do první poloviny prvního kvadrantu. Funkce jsou periodické po periodě 2π, proto je možné převést x do množiny $< 0; 2\pi)$. Následuje převod do prvního kvadrantu podle vzorce (6.1) pro sin, a (6.4) pro cos. Dále je proveden převod do první poloviny kvadrantu podle vzorce (6.2) pro sin, a (6.5) pro cos. Nakonec jsou vyčísleny funkce $sin(x)$(6.3) a $cos(x)$(6.6).

$$sin(x) = \begin{cases} sin(x) & \text{když } \leq x \leq \frac{\pi}{2} \\ sin(-x) & \text{když } \frac{\pi}{2} \leq x \leq \pi \\ -sin(x) & \text{když } \pi \leq x \leq \frac{3\pi}{2} \\ -sin(-x) & \text{když } \frac{3\pi}{2} \leq x \leq 2\pi \end{cases} \tag{6.1}$$

$$sin(x) = \begin{cases} sin(x) & \text{když } 0 \leq x \leq \frac{\pi}{4} \\ cos(\frac{\pi}{2} - x) & \text{když } \frac{\pi}{4} \leq x \leq \frac{\pi}{2} \end{cases} \tag{6.2}$$

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} \tag{6.3}$$

$$cos(x) = \begin{cases} \cos(x) & \text{když } \leq x \leq \frac{\pi}{2} \\ -\cos(-x) & \text{když } \frac{\pi}{2} \leq x \leq \pi \\ -\cos(x) & \text{když } \pi \leq x \leq \frac{3\pi}{2} \\ \cos(-x) & \text{když } \frac{3\pi}{2} \leq x \leq 2\pi \end{cases} \tag{6.4}$$

$$cos(x) = \begin{cases} \cos(x) & \text{když } 0 \leq x \leq \frac{\pi}{4} \\ \sin(\frac{\pi}{2} - x) & \text{když } \frac{\pi}{4} \leq x \leq \frac{\pi}{2} \end{cases} \tag{6.5}$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \tag{6.6}$$

Funkce e^x je implementována podle vzorce (6.7).

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} \tag{6.7}$$

6.3 Spuštění programu

V první řadě je nutné před spuštěním projekt přeložit příkazem "*make*"v adresáři *src*. V adresáři *src/src/test* se nachází některé programy, které byly využity v experimentech. Programy je možné přeložit pomocí příkazu "*make test*". Testovací programy se nachází v adresáři *src/src/test*.

Aplikace se spouští příkazem *src/debug k*, kde *k* je celé číslo, a označuje již předpřipravený problém výpočtu integrálu ve vnitřně interpretovatelných instrukcí. Níže v tabulce jsou uvedeny možné parametry. Číslo *k* v levém sloupci a předpřipravený integrál v pravém sloupci. Implementace neobsahuje modul pro překlad integrálu popsaného v textové podobě do integrálu popsaného v interpretovatelných instrukcí. V budoucnu je možné využít *yaml* a *bison* pro překlad problému do navržených vnitřních instrukcí.

1	$\int_0^{10} sin(x) * cos(x)dx$
2	$\int_0^{10} cos(x)dx$
3	$\int_0^{16} e^x dx$
4	$\int_{10}^{26} \int_0^{y} \frac{x^2}{y} dxdy$
5	$\int_{10}^{14} \int_{-4}^{4} e^{\frac{x}{y}} dxdy$

Tabulka 6.1: Seznam možných parametrů

Po startu aplikace se spustí inicializace problému. V příloze je ukázka funkce pro inicializaci jednoho z konkrétních integrálu použitého v experimentech. Do příslušných proměnných je uložen daný problém v podobě virtuálních instrukcí. Pro každý rozměr integrálu je potřeba vytvořit funkci, která se zavolá před výpočtem daného rozměru a nastaví počáteční a koncový bod integrace. Následně jsou inicializovány moduly *int_math*a *program_stack* . Pak je spuštěn výpočet inverzní matice. Nakonec je spuštěna funkce solve, která rekurzivně řeší daný integrál.

Kapitola 7

Experimenty v SW

Softwarové experimenty probíhaly na PC *gigabyte 965p-ds3* s CPU *intel core 2 duo* s 8GB ram. Pro experimenty byly použity moduly s implementací inicializace požadovaných problémů. Pro některé experimenty byly požity programy uložené v adresáři *src/src/test* Slouží například pro výpočet a výpis derivací pomocí numerického řešení inverzní matice 3.7.1. Program byl ověřován pro různé kroky, různě velké inverzní matice, a různě velké datové typy.

7.1 Matematické funkce

Jako první jsem testoval přesnost výpočtů implementovaných matematických funkcí $sin(x)$, $cos(x)$, e^x. V tabulce 7.1 je uvedena přesnost, rychlost, a orientačně výsledek výpočtu. Celý předpokládaný výsledek by zabíral příliš mnoho místa. Všechny výsledky se shodují od začátku na více jak 1800 platných dekadických míst.

Function	**doba běhu**	**výsledek**	**rozdíl**
sin(12)	0.012177s	-0.536572918	0.16510358e-2466
cos(12)	0.012056s	0.843853958	0.10498275e-2466
exp(12)	0.000101s	162754.7914	0.14751490e-1817
sin(0.125)	0.009979s	0.124674733	0.94362691e-2483
cos(0.125)	0.009900s	0.992197667	0.89533474e-2483
exp(0.125)	0.002463s	1.133148453	0.96358385e-1823
sin(1000)	0.012169s	0.826879540	0.17495061e-2464
cos(1000)	0.012241s	0.562379076	0.25723411e-2464
exp(1000)	0.000273s	1.970071114e431	0.26402525e-1382
sin(-100000)	0.008835s	-0.035748797	0.31120419e-2462
cos(-100000)	0.008681s	-0.999360807	0.11132291e-2463
exp(-100000)	0.000412s	3.562949565e-43430	0.12210180e-45244

Tabulka 7.1: přesnost výpočtu funkcí sin(x) a cos(x) s typem gmp-8192

7.2 Rychlost výpočtu Taylorova polynomu

V této sekci jsem testoval závislost počtu potřebných derivací na velikosti kroku. Přitom byla zkoumána i rychlost výpočtu v podobě celkového počtu potřebných sčítání. Test byl proveden na funkcích $sin(x)$ a e^x. Derivace byly vypočítány pomocí vzorce (7.1) pro $sin(x)$ a pomocí vzorce (7.2) pro e^x.

$$sin^{(i)}(x) = \begin{cases} sin(x) & \text{když i mod } 4 = 0 \\ cos(x) & \text{když i mod } 4 = 1 \\ -sin(x) & \text{když i mod } 4 = 2 \\ -cos(x) & \text{když i mod } 4 = 3 \end{cases} \tag{7.1}$$

$$exp(x)(i) = e^x \tag{7.2}$$

V první tabulce 7.2 je znázorněn výpočet hodnoty $sin(x)$, pro $x = 100$ s počátkem v bodě $x = 0$. V druhé tabulce 7.3 je znázorněn výpočet hodnoty e^x, pro x = 100 s počátkem aproximace $x = 0$. V obou případech byly všechny výsledky i mezivýsledky naprosto přesné pro zadanou velikost aritmetiky. Výpočet probíhal s datovým typem o velikosti *gmp-16384*.

Velikost kroku	**počet kroků**	**Minimální počet derivací na krok**	**Maximální počet derivací na krok**	**celkový počet sčítání**
0.125	800	1378	1379	1 103 118
0.25	400	1490	1491	596 064
0.5	200	1618	1619	323 664
1	100	1768	1770	176 863
10	10	2526	2528	25 272
50	2	3518	3518	7 036
100	1	4186	4186	4 186

Tabulka 7.2: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou z uzlových bodů

Velikost kroku	**počet kroků**	**Minimální počet derivací na krok**	**Maximální počet derivací na krok**	**celkový počet sčítání**
0.125	800	1374	1369	1 097 059
0.25	400	1479	1484	592 444
0.5	200	1606	1612	321 742
1	100	1756	1762	175 862
10	10	2509	2516	25 119
50	2	3487	3488	6 975
100	1	4138	4138	4 138

Tabulka 7.3: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou z uzlových bodů

7.3 Výpočet derivací

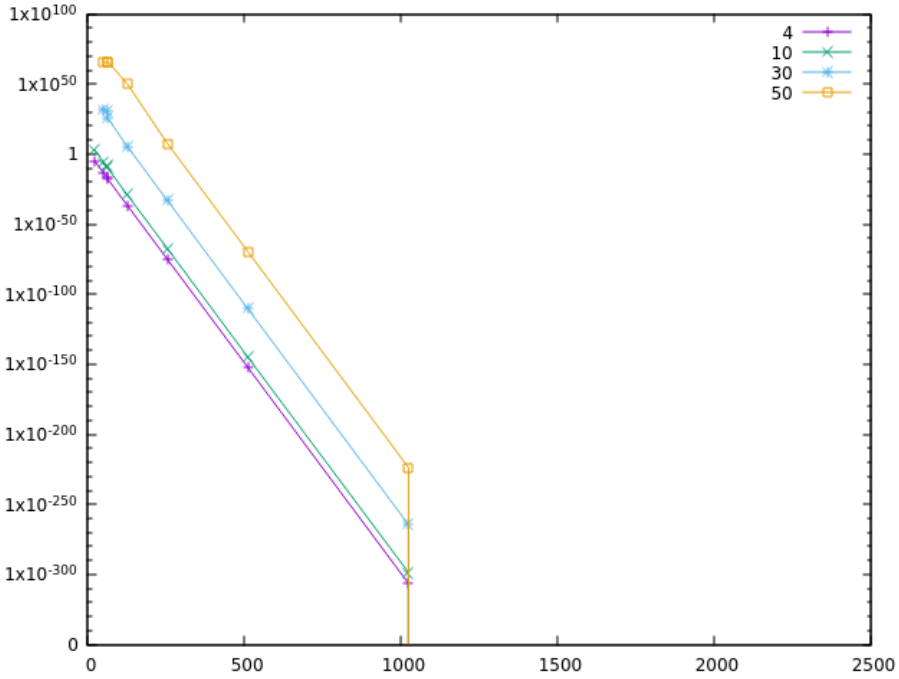
V těchto experimentech zjišťuji závislost chyby výpočtu vyšších derivací v bodě x_k na velikosti kroku pro výpočet prvních derivací, počtu prvků v inverzní matici a velikostí datového typu. Čísla v tabulkách jsou velice velká a mají mnoho platných číslic. Pro zobrazení čísel v požadované délce byla využita metoda ořezání. V první řadě je vhodné ověřit přesnost inverzní matice. Přesnost inverzní matice jsem ověřoval pomocí jednotkové matice. Nejdříve se sestaví jednotková matice. Následně je Gaussovou eliminací vypočtena inverzní matice. Po roznásobení matice a inverzní matice by měla vzniknout jednotková matice. Porovnáním jednotkové matice a výsledné matice dostaneme požadovanou chybu.

Prvním experiment je zaměřen na přesnost výpočtu inverzní matice pro dopřednou diferenci. V několika experimentech jsem zjistil, že pro ostatní diference je maximální chyba v inverzní matici menší. V tabulce 7.4 je udaná maximální odchylka výsledné matice od jednotkové matice. V prvním sloupci tabulky je datový typ použitý pro výpočet inverzní matice. Číslo za pomlčkou znázorňuje velikost desetinné části datového typu v bitech. V hlavičce tabulky je uveden rozměr inverzní matice představující počet použitých okolních bodů pro aproximaci derivací. Ve sloupcích je uvedena maximální odchylka od jednotkové matice vztažena k odpovídajícímu datovému typu a počtu okolích bodů. Pro aproximaci ze čtyř okolních bodů je potřeba znát pět bodů. Čtyři okolní body plus bod ve kterém chceme znát derivace vyšších řádů. jak je vidět v tabulce níže pro datový typ *gmp-2048* je chyba výpočtu inverzní matice relativně malá.

Type	**4**	**10**	**30**	**50**
float	1.52e-05	2.04e+03	nan	nan
double	-2.84e-14	-1.90e-06	-3.31e+31	5.75e+65
long-double	1.38e-17	-2.32e-09	1.91e+30	1.79e+65
gmp-64	2.92e-18	-1.73e-10	7.36e+25	1.55e+66
gmp-128	1.58e-37	5.79e-30	-1.60e+05	-8.07e+49
gmp-256	4.66e-76	-2.75e-68	4.81e-34	-2.74e+07
gmp-512	4.02e-153	1.03-145	-3.38e-111	2.20e-70
gmp-1024	3.00e-307	-1.77e-299	2.96e-265	-1.71e-224
gmp-2048	0.00e+00	0.00e+00	0.00e+00	0.00e+00

Tabulka 7.4: Chyba výpočtu inverzní matice

Obrázek 7.1 zobrazuje klesající absolutní hodnotu chyby výpočtu inverzní matice v závislosti na rostoucí velikosti datového typu s plovoucí řádovou čárou. Zajímavé je, že chyba klesá geometricky nikoliv lineárně. Na logaritmické y-ové ose je zobrazena chyba ve výpočtu. Na x-ové ose je zobrazena přesnost. Jednotlivé funkce v grafu představují velikost inverzní matice vypočítané pomocí rovnic pro dopřednou differenci.



Obrázek 7.1: přesnost výpočtu inverzní matice

V následujícím experimentu jsem zjišťoval přesnost výpočtu derivací na velikosti datového typu. Pro výpočet byla využita inverzní matice pro aproximaci 50 derivací pomocí kombinované diference. V tabulce 7.5 je zobrazena přesnost výpočtu některých derivací. Přesnosti vypočtu derivací byla měřena pomocí funkce $sin(x)$, která ma nekonečný počet derivací. Funkce sinus je systém, který se nachází na hranici stability. Tabulka ukazuje rozdíl mezi očekávaným výsledkem a získaným výsledkem. V prvním sloupci se nachází použitý datový typ. Přičemž gmp-x značí datový typ *mpf_t* z knihovny gmp s minimální přesností x bitů. Ve druhém sloupci je krok.

V tabulkách 7.6, 7.7 a 7.8 je zobrazena chyba několika sčítanců Taylorovy řady $DX(i) = \frac{f^{(i)(x_k)}}{i!}(h^i)$.

$$f(x_k + h) = f(x_k) + DX1 + DX2 + ... + DXN$$

$$DX(i) = \frac{f^{(i)}(x)}{i!}(h^i)$$

Tato chyba předurčuje přibližnou velikost integračního kroku. Většinou je integrační krok větší než derivační. Každý prvek se při následném integrování násobí koeficientem $\frac{k^{i+1}}{i*h^i}$, kde k je integrační krok, h je derivační krok a i je derivace. V tabulkách jsou použity kroky $h = 1, 0.125$ a 0.015625. V tabulce 7.5 jsou sudé derivace daleko přesnější jako liché, to může být způsobeno vlastností funkce $sin^{(i)}(x) = 0|i$ je sudé

Type	h	1. der	4. der	7. der	10. der
$sin^{(i)}(0)$	-	1	0	-1	0
float	1	2.58e-04	-7.63e-05	-9.60e-02	3.24e-03
float	0.125	2.26e-06	-1.55e-02	-1.61e+02	3.79e+05
double	1	2.53e-04	-7.32e-15	-9.55e-02	3.01e-13
double	0.125	3.27e-13	5.04e-12	-3.13e-05	-1.99e-04
double	0.015625	-7.10e-15	2.78e-09	2.15e-02	-2.86e+04
long-double	1	2.53e-04	-6.75e-17	-9.55e-02	2.89e-15
long-double	0.125	3.34e-13	-3.13e-15	-3.13e-05	2.88e-07
long-double	0.015625	1.20e-17	1.93e-11	-4.35e-05	9.76e+00
gmp-128	1	2.53e-04	-1.08e-38	-9.55e-02	2.78e-37
gmp-128	0.125	3.34e-13	-3.66e-35	-3.13e-05	1.58e-28
gmp-128	0.015625	3.12e-22	-9.62e-32	-7.69e-09	1.26e-19
gmp-512	1	2.53e-04	-2.13e-154	-9.55e-02	5.74e-153
gmp-512	0.125	3.34e-13	-6.56e-151	-3.13e-05	2.60e-144
gmp-512	0.015625	3.12e-22	-1.32e-147	-7.69e-09	1.68e-135
gmp-2048	1	2.53e-04	0.00e+00	-9.55e-02	0.00e+00
gmp-2048	0.125	3.34e-13	0.00e+00	-3.13e-05	0.00e+00
gmp-2048	0.015625	3.12e-22	0.00e+00	-7.69e-09	0.00e+00
gmp-2048	0.000976562	2.84e-34	0.00e+00	-1.17e-13	0.00e+00

Tabulka 7.5: Chyba derivace 1.,4.,7. a 10. derivace pro funkci $sin(0)$

Type	1. člen	4. člen	7. člen	10. člen
Analyticky	0.5	0	-0.248...e-04	0
float	1.29e-04	-6.35e-07	-2.38e-06	8.11e-11
long-double	1.26e-04	-5.63e-19	-2.36e-06	7.25e-23
gmp-512	1.26e-04	-1.77e-156	-2.36e-06	1.43e-160
gmp-2048	1.26e-04	0.00e+00	-2.36e-06	0.00e+00

Tabulka 7.6: Chyba jednotlivých členů Taylorova polynomu pro krok h=1

Type	1. člen	4. člen	7. člen	10. člen
Analyticky	0.0078125	0	-0.147...e-11	0
float	1.76e-08	-3.95e-09	-2.38e-10	1.10e-12
long-double	2.61e-15	-7.98e-22	-4.63e-17	8.39e-25
gmp-512	2.61e-15	-1.67e-157	-4.63e-17	7.58e-162
gmp-2048	2.61e-15	0.00e+00	-4.63e-17	0.00e+00

Tabulka 7.7: Chyba jednotlivých členů Taylorova polynomu pro krok h=0.125

Type	1. člen	4. člen	7. člen	10. člen
Analyticky	0.122...e-03	0	-0.9...e-19	0
float	-8.00e-10	-2.30e-10	-3.20e-12	2.70e-14
long-double	1.47e-21	1.50e-22	-3.83e-24	3.31e-27
gmp-512	3.81e-26	-1.02e-158	-6.78e-28	5.72e-163
gmp-2048	3.81e-26	0.00e+00	-6.78e-28	0.00e+00

Tabulka 7.8: Chyba jednotlivých členů Taylorova polynomu pro krok h=0.015625

V následujících tabulkách jsou uvedeny chyby ve výpočtu derivací pro různě velké inverzní matice. Derivace byly počítány pro funkci $sin(x)$ v bodě 0. Datový typ byl použit gmp-2048. Tabulky ukazují postupně použité velikost kroku $h = 0.125$, $h = 0.015625$, a $h = 0.001953125$.

der.	ocek.	20	30	40	50	60	70	80
1. der.	1	2.2e-25	1.6e-37	1.2e-49	1.0e-61	8.7e-74	7.2e-86	6.1e-98
9. der.	1	1.5e-14	1.5e-26	1.4e-38	1.2e-50	1.0e-62	9.3e-75	8.1e-87
15. der.	-1	5.1e-07	1.2e-18	1.5e-30	1.6e-42	1.6e-54	1.5e-66	1.4e-78
29. der.	1	-	2.0e-02	6.5e-13	2.4e-24	4.9e-36	7.3e-48	9.2e-60
39. der.	-1	-	-	2.6e-02	2.2e-12	1.5e-23	4.8e-35	9.7e-47
49. der.	1	-	-	-	3.3e-02	6.3e-12	7.8e-23	3.5e-34
59. der.	1	-	-	-	-	3.9e-02	1.5e-11	3.1e-22
69. derivace	1	-	-	-	-	-	4.5e-02	3.3e-11

Tabulka 7.9: Chyba některých derivací v závislosti na velikosti inverzní matice a kroku h=0.125

der.	ocek.	20	30	40	50	60	70	80
1. der.	1	1.9e-43	1.3e-64	1.0e-85	7.6e-107	5.8e-128	4.6e-149	3.6e-170
9. der.	1	2.2e-25	2.1e-46	1.8e-67	1.4e-88	1.2e-109	9.9e-131	8.1e-152
15. der.	-1	1.9e-12	4.4e-33	5.2e-54	5.2e-75	4.8e-96	4.2e-117	3.6e-138
29. der.	1	-	3.2e-04	9.7e-24	3.4e-44	6.4e-65	8.9e-86	1.0e-106
39. der.	-1	-	-	4.2e-04	3.4e-23	2.2e-43	6.3e-64	1.1e-84
49. der.	1	-	-	-	5.2e-04	9.6e-23	1.1e-42	4.6e-63
59. der.	1	-	-	-	-	6.3e-04	2.3e-22	4.4e-42
69. der.	1	-	-	-	-	-	7.3e-04	5.0e-22

Tabulka 7.10: Chyba některých derivací v závislosti na velikosti inverzní matice a kroku h=0.015625

der.	ocek.	20	30	40	50	60	70	80
1. der.	1	1.6e-61	1.0e-91	7.5e-122	5.3e-152	3.8e-182	2.8e-212	2.0e-242
9. der.	1	3.3e-36	2.9e-66	2.2e-96	1.7e-126	1.3e-156	1.0e-186	7.7e-217
15. der.	-1	3.4e-06	2.8e-35	1.7e-77	5.6e-95	1.3e-137	1.1e-167	-9.1e-198
29. der.	1	-	5.0e-06	1.4e-34	4.6e-64	8.1e-94	1.0e-123	1.1e-153
39. der.	-1	-	-	6.6e-06	4.9e-34	3.0e-63	8.0e-93	1.4e-122
49. der.	1	-	-	-	8.2e-06	1.4e-33	1.4e-62	5.9e-92
59. der.	1	-	-	-	-	9.8e-06	3.3e-33	6.0e-62
69. der.	1	-	-	-	-	-	1.1e-05	7.3e-33

Tabulka 7.11: Chyba některých derivací v závislosti na velikosti inverzní matice a kroku h=0.001953125

Z tabulek je zřejmé, že se zvětšující velikostí inverzním matice se zvyšuje přesnost jednotlivých derivací lineárně. Zvětšování inverzní matice do nekonečna není možné. Pro inverzní matici s 200 prvky je chyba první derivace $2.474040e-01$. Z toho je možné vyvodit závěr, že zvětšování je omezeno počtem platných čísel v datového typu. Stejně tak zmenšování kroku je omezeno datovým typem.

7.4 Integrace

V následujících experimentech zjišťuji závislost chyby výpočtu na velikosti kroku integrace a vzdálenosti uzlových bodů aproximace (velikost derivačního kroku). Přesnost integrace není zjišťována jen pro jednoduchý integrál, ale také pro integrály vyšších řádů. Pro experimenty jsou použil integrály z literatury [6] a [12]. V tabulkách je v prvním řádku vždy zobrazen předpokládaný výsledek vypočítaný analyticky. V prvním sloupci je vždy zobrazena velikost inverzní matice.

Tabulka 7.12 pro integrál:

$$\int_0^k cos(x)dx = sin(k) - sin(0)$$

počet der	**der krok**	**int krok**	$k = 2$	$k = 4$	$k = 8$	$k = 16$
anal.	-	-	0.9092974	-0.7568024	0.98935824	-0.2879033
10	0.125	1.0	0.573e-09	0.113e-08	0.399e-09	0.118e-08
10	0.125	0.5	0.228e-13	0.338e-13	0.174e-13	0.376e-13
10	0.125	0.25	0.271e-14	0.349e-14	0.215e-14	0.402e-14
10	0.015625	1.0	0.196e-08	0.388e-08	0.136e-08	0.406e-08
10	0.015625	0.5	0.115e-11	0.171e-11	0.879e-12	0.190e-11
10	0.015625	0.25	0.500e-15	0.657e-15	0.394e-15	0.754e-15
20	0.125	1.0	0.220e-24	0.460e-24	0.150e-24	0.477e-24
20	0.125	0.5	0.286e-26	0.430e-26	0.217e-26	0.477e-26
20	0.125	0.25	0.816e-27	0.105e-26	0.646e-27	0.121e-26
20	0.015625	1.0	-0.726e-21	-0.151e-20	-0.494e-21	-0.157e-20
20	0.015625	0.5	-0.327e-27	-0.497e-27	-0.249e-27	-0.550e-27
20	0.015625	0.25	0.363e-43	0.451e-43	0.289e-43	0.525e-43
40	0.125	1.0	0.395e-50	0.853e-50	0.265e-50	0.879e-50
40	0.125	0.5	0.560e-51	0.850e-51	0.425e-51	0.941e-51
40	0.125	0.25	0.228e-51	0.296e-51	0.181e-51	0.340e-51
40	0.015625	1.0	-0.291e-51	-0.630e-51	-0.195e-51	-0.649e-51
40	0.015625	0.5	-0.125e-64	-0.192e-64	-0.948e-65	-0.212e-64
40	0.015625	0.25	0.135e-82	0.180e-82	0.106e-82	0.206e-82
80	0.125	1.00	0.322e-99	0.705e-99	0.214e-99	0.725e-99
80	0.125	0.5	0.110e-99	0.168e-99	0.837e-100	0.185e-99
80	0.125	0.25	0.532e-100	0.688e-100	0.420e-100	0.792e-100
80	0.015625	1.0	-0.304e-125	-0.671e-125	-0.202e-125	-0.6893968e-125
80	0.015625	0.5	0.538e-161	0.833e-161	0.4067e-161	0.9180e-161
80	0.015625	0.25	0.290e-170	0.388e-170	0.228e-170	0.4438e-170

Tabulka 7.12: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou

Tabulka pro integrál:

$$\int_0^k e^x dx = e^k - e^0$$

počet der	**der krok**	**int krok**	$k = 2$	$k = 4$	$k = 8$	$k = 16$
anal.	-	-	6.389056	53.59815	2979.9579	8886109.52
10	0.125	1.0	0.244e-8	0.205e-7	0.113e-5	0.339e-2
10	0.125	0.25	0.120e-13	0.100e-12	0.560e-11	0.167e-7
10	0.015625	1.0	0.827e-8	0.693e-7	0.693e-7	0.115e-1
10	0.015625	0.25	0.218e-14	0.183e-13	0.101e-11	0.303e-8
20	0.015625	0.25	0.165e-33	0.138e-32	0.769e-31	0.229e-27
40	0.015625	0.25	-0.588e-82	-0.493e-81	-0.274e-79	-0.818e-76
80	0.015625	0.25	-0.126e-169	-0.106-168	-0.589e-167	-0.175e-163

Tabulka 7.13: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou

V tabulce 7.14 je zobrazen výsledek integrálu zobrazeného před tabulkou s derivačním krokem $step\_diff_1 = 0.5$, a integračním krokem $step\_int_1 = 2.0$ pro proměnou y, a derivačním krokem $step\_diff_0 = 0.5$, a integračním krokem $step\_int_0 = 2.0$ pro proměnou x

$$\int_{y_0}^{y_1}\int_{0}^{y}\frac{x^2}{y}dxdy = \int_{y_0}^{y_1}\left[\frac{1}{3}*\frac{x^3}{y}\right]_0^y dy = \int_{y_0}^{y_1}\frac{1}{3}*\left(\frac{y^3}{y}-\frac{0}{y}\right)dy = \frac{1}{9}(y_1^3 - y_0^3)$$
$$y_0 = 10$$

počet	$y_1 = 12$	$y_1 = 14$	$y_1 = 18$	$y_1 = 26$
analyticky	80.888888	193.777777	536.8888888	1841.7778
10	0.9361e-2464	0.2053e-2463	0.5460e-2463	0.1530e-2462

Tabulka 7.14: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou. $step\_der_1 = 0.5$ $step\_int_1 = 2$

Pro další experiment jsem zvolil integrál (7.3), výsledky experimentu jsou zobrazeny v několika tabulkách. Referenční hodnoty byly vypočítány pomocí webového programu wolfram alpha. Pro první tabulku 7.15 jsem zvolil derivační krok $step\_der_1 = 0.125$ a integrační krok $step\_int_1 = 2.0$ pro proměnou y. V tabulce 7.16 jsem zvolil derivační krok $step\_der_1 = 0.125$ a integrační krok $step\_int_1 = 0.5$ pro proměnou y. V obou tabulkách $step\_der_0$ a $step\_int_0$ představují derivační a integrační krok pro proměnou x.

$$\int_{10}^{k}\int_{-4}^{4} e^{\frac{x}{y}}dxdy \tag{7.3}$$

počet	$step\_der_0$	$step\_int_0$	$k = 12$	$k = 14$	$k = 18$	$k = 26$
anal.	-	-	16.357960	32.61315	64.952873	129.318198
10	0.125	1.0	0.701e-8	0.764562e-8	0.774333e-8	0.774753e-8
10	0.125	0.25	0.701e-8	0.764562e-8	0.774333e-8	0.774753e-8
10	0.015625	1.00	0.701e-8	0.764e-8	0.774e-8	0.774e-8
10	0.015625	0.25	0.701e-8	0.764e-8	0.774e-8	0.774e-8
20	0.015625	1.00	0.208e-15	0.211e-15	0.211e-15	0.211e-15
20	0.015625	0.25	0.208e-15	0.211e-15	0.211e-15	0.211e-15
40	0.015625	0.50	0.431e-35	0.431e-35	0.431e-35	0.431e-35
40	0.015625	0.25	0.431e-35	0.431e-35	0.431e-35	0.431e-35
80	0.015625	1.00	0.108e-52	0.108e-52	0.108e-52	0.108e-52
80	0.015625	0.25	0.108e-52	0.108e-52	0.108e-52	0.108e-52

Tabulka 7.15: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou

počet	$step\_der_0$	$step\_int_0$	$k = 12$	$k = 14$	$k = 18$	$k = 26$
anal.	-	-	16.357960	32.61315	64.952873	129.318198
10	0.125	1.0	0.227e-16	0.249e-16	0.252e-16	0.252e-16
10	0.125	0.25	0.227e-16	0.249e-16	0.253e-16	0.253e-16
10	0.015625	1.00	0.226e-16	0.248e-16	0.251e-16	0.251e-16
10	0.015625	0.25	0.227e-16	0.249e-16	0.253e-16	0.253e-16
20	0.015625	1.00	0.773e-27	0.784e-27	0.784e-27	0.784e-27
20	0.015625	0.25	0.773e-27	0.784e-27	0.784e-27	0.784e-27
40	0.015625	2.00	0.580e-41	0.580e-41	0.580e-41	0.580e-41
40	0.015625	1.00	0.580e-41	0.580e-41	0.580e-41	0.580e-41
80	0.015625	4.00	0.580e-41	0.580e-41	0.580e-41	0.580e-41

Tabulka 7.16: rozdíl získané a požadované hodnoty při výpočtu integrálu Taylorovou metodou

Kapitola 8

Implementace v HW

V dnešní době je zvyšování výkonu bez využití paralelizmu velice těžké. Křemíkové čipy začínají narážet na své fyzikální limity a objevování nových algoritmů je nepravděpodobné. Proto se zejména v dnešní době začíná využívat masivní paralelizmus grafických karet (GPUGP), ASIC čipů nebo reprogramovatelných obvodů FPGA. Velkou výhodou vestavěných systémů je vysoký výkon a minimální počet zdrojů (velikost řídící jednotky). Nevýhodou je specifická aplikace, která nelze nikde jinde použít.

8.1 FPGA

Dnes existují dva majoritní jazyky pro vyjádření funkcionality navrhovaného obvodu. Prvním jazykem je Verilog využívaný především ve Spojených státech amerických. Druhým jazykem je VHDL využívaný hlavně v Evropě. Pro vytváření verifikačního prostředí se využívá jazyk SystemVerilog, který je nadstavbou nad Verilogem. SystemVerilog se využívá jak v kombinaci s Verilogem, tak i pro verifikaci obvodů popsaných jazykem VHDL.

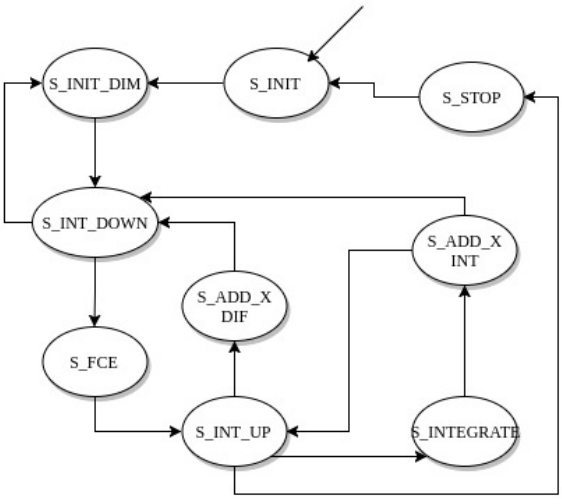
Překlad z popisu obvodu do FPGA nebo ASIC čipů probíhá v několika fázích. Prví fází je syntéza, pří které probíhá překlad z popisu daného obvodu do reprezentace komponent *and grafem*. Druhou důležitou fází je fáze place and route. V této fázi dochází k mapování *and grafu* do cílové architektury.

Základem FPGA jsou *registry* a *look up table* (LUT). LUT reprezentují n-rozměrnou funkci $y = f(x_0, x_1, ..., x_{n-1})$. Definiční obor jednotlivých rozměrů funkce je zbytková třída dvou, tedy $x_i \in \{0; 1\}$. Obor hodnot se taktéž nachází ve zbytkové třídě dvou, tedy $y \in \{0; 1\}$. LUT je hardwarově implementována jako malá paměť, pak lze vstup funkce interpretovat jako adresu $x_0x_1...x_{n-1}$ zapsanou ve dvojkové soustavě. Výsledná hodnota y je binární hodnota uložena na dané adrese. Kombinací paralelního a sériového zapojení lze z více LUT vytvořit téměř libovolné funkce. Obvod se také skládá z registrů, které jsou umístěny mezi LUT a slouží k uložení mezivýsledku. Pomocí registrů se zkracují kritické cesty, které zhoršují časování obvodu. Potřeba průmyslu došla do bodu, kdy již nestačí používat v FPGA pouze registry a LUT. Obvody sestavené pouze z těchto komponent by zabíraly příliš mnoho zdrojů a nevešly by se do FPGA, nebo by nebyly schopné splnit časování. Proto se v dnešní době integrují do FPGA RAM paměti, DSP procesory, posuvné registry, a jiné komponenty.

8.2 Návrh

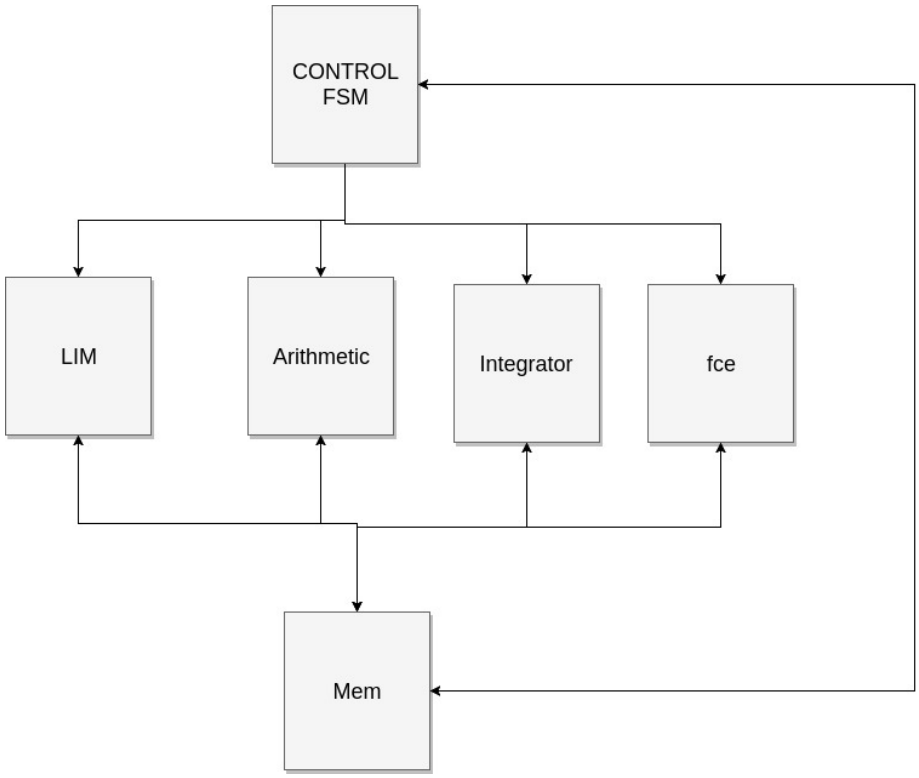
Pro implementaci jsem využil jazyk VHDL-2008. Tento jazyk jsem zvolil, protože umožňuje vytvářet pole, jejichž velikost je specifikována generickými parametry v entitě. Možnost pracovat s poli ušetří při implementaci nemálo času. Nebylo potřeba některé vstupy do entit serializovat a následně deserializovat. Návrh systému je vyobrazen na obrázku 8.2. Snažil jsem provést co možná nejvhodnější návrh pro pozdější změny. Většinu funkcionality jsem proto rozdělit od entit. Systém obsahuje stavový stroj zobrazený na obrázku 8.1. V modulu *type.vhd* je implementován datový typ. Jsou zde implementovány operace sčítání, odčítání, násobení, a děleni. Hodnoty potřebné pro integrování se ukládají do paměti. Zajímavým vylepšením by bylo sloučit komponenty *LIM*, *FCE*, a *integrator*. Z těchto komponent je v jeden časový okamžik vždy použita jenom jedna. Komponenty lze sloučit a jednotlivé funkce vykonávat mikrokódem.

Výpočet začne přechodem do stavu *S_INIT*. Následně se systém dostane do stavu *S_INIT_DIM*, ve kterém se inicializují všechny úrovně integrálu. Dojde k inicializaci hodnot x a x_end. Ve stavu se pomocí komponenty *LIM* inicializují meze na začátku výpočtu aktuální úrovně integrálu. Následuje přechod do stavu *S_INT_DOWN*, kde se hodnota x uloží do proměnné x_save. V tomto stavu se rozhoduje, zda se bude počítat nižší integrál nebo dojde k vyčíslení funkce. Ve stavu *S_FCE* se vyčísluje nejnižší integrovaná funkce. Ze stavu se přechází do stavu *S_INT_UP* po dokončení vyčíslení funkce. ve stavu *S_INT_UP* se rozhodne podle proměnných do kterého stavu se přechází. Nejčastěji se přechází do stavu *S_ADD_X_DIF*, ve kterém se přičte k x hodnota diferenčního kroku. Pro přechod do stavu *S_Integrate*, musí být již spočítáno potřebný počet derivací. Ve stavu dojde k numerické integraci pomocí komponenty *Integrator*. Ve stavu *S_ADD_X_INT* dojde k posuvu hodnoty x o integrační krok. Ve stavu *S_ADD_X_INT* dochází k rozhodnutí zda se má počítat nižší integrál nebo dojde k ukončení výpočtu stávajícího integrálu a přechodu do stavu *S_INT_UP*. K tomuto přechodu dojde, pokud $x = x\_end$. K ukončení výpočtu dojde, pokud ve stavu *S_INT_UP* je nastavena hodnota uložená v konstantě *dim*. Konstanta *dim* představuje počet dimenzí v integrálu.



Obrázek 8.1: Stavový automat system

Implementace integračního systému obsahuje několik komponent. Komponenty *CONTROL FSM* a *MEM* nejsou implementovány jako podkomponenty. Komponenta *CONTROL FSM* řídí celý chod integrovacího systému. Komponenta *LIM* se stará o vypočítání krajních hodnot integrálu. V komponentě může být implementován výpočet funkce. Komponenta *ARITHMETIC* slouží k přičítání kroků k hodnotě x. Komponenta *INTEGRATOR* implementuje integrování pomocí Taylorova rozvoje. Poslední komponentou je *FCE*, která slouží k implementaci primitivní funkce. Pro každý specifický integrál je potřeba změnit hodnotu konstanty *dimension* a implementaci podkomponenty *LIM* a podkomponenty *FCE*.



Obrázek 8.2: system.vhd

Rozhraní komponent LIM FCE a jak mají komunikovat s okolním prostředím.

8.3 Datový typ

Velikost datového typu lze změnit v souboru *type_vivado.vhd* pro překlad a v souboru *type_sim.vhd* pro simulaci. Pro jednoduchost implementace jsem zvolil datový typ s pevnou desetinnou čárkou. Všechny ostatní moduly využívají sčítačku, odčítačku, násobičku, a děličku z tohoto modulu. Je tedy jednoduché změnit datový typ. Stačí změnit implementaci v modulu *type.vhd* a celý systém bude používat po překladu nový datový typ. Implementovaný datový typ s pevnou desetinnou čárkou má několik parametrů, kterými je možné měnit velikosti jednotlivých částí datového typu. Prvním parametrem je *my_type_size*, který určuje celkový počet platných číslic ve dvojkové soustavě. Druhým důležitým parametrem je *my_type_decimal_point*, který určuje počet platných číslic za desetinnou čárkou.

Každá z jednotek má stejné rozhraní. Rozhraní zobrazené níže je rozhraním sčítačky. Prvními dvěma vstupy jsou hodiny a reset. Pro funkci obvodu musí být reset nastaven do

nuly. Rozhraní obsahuje jednoduchý komunikační protokol nacházející se na vstupu a výstupu obvodu. Komunikační protokol obsahuje dva signály **SRC_RDY** a **DST_RDY**. **SRC_RDY** představuje připravenost zdroje předat data, a **DST_RDY** představuje připravenost cíle přebrat data. Hodnoty jsou aktivní v jedničce. K předáním hodnot dojde, pokud jsou obě hodnoty nastaveny v jedničce. Výpočet probíhá ve třech krocích.

1. Předání vstupu (IN_SRC_RDY = '1' a IN_DST_RDY = '1')
2. Výpočet
3. Předání výstupu (OUT_SRC_RDY = '1' a OUT_DST_RDY = '1')

Ze všech čtyř jednotek je sestavena jedna jednotka arithmetic, která dokáže provádět vždy jednu ze čtyřech operací. Navíc má pouze vstup určující operaci, jinak její rozhraní má totožné chování s rozhraním jednotek pro provádění operací.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.type_pack.all;



entity add is
port (
        CLK                 : in  std_logic;
        RESET               : in  std_logic;
        -- ------------------------------------
        IN_SRC_RDY          : in  std_logic;
        IN_DST_RDY          : out std_logic;
        IN_DATA_1           : in my_type;
        IN_DATA_2           : in my_type;

        OUT_SRC_RDY         : out std_logic;
        OUT_DST_RDY         : in  std_logic;
        OUT_DATA            : out my_type);
end entity add;
```

8.3.1 Sčítání a odčítání

Sčítání a odčítání je implementováno vestavěným algoritmem pro klasický celočíselný typ. Zde se nemusí tvořit speciální úpravy pro datový typ s pevnou desetinnou čárkou. Výsledek je v taktu, ve kterém jsou data přivedena na vstup obvodu. Jedná se tedy pouze o kombinační logiku bez registrů.

8.3.2 Násobení

Pro násobení jsem zvolil Boothův algoritmus násobení s radixem 2 [4]. Boothův algoritmus je založen na dílčích součtech a posuvech. V každém kroku se rozhodne podle dvou nejvýznamnějších bitů zda se bude násobitel přičítat, odčítat nebo se přičtou samé nuly.

Operace jsou zobrazeny v tabulce 8.1. Na konci kroku se výsledek vždy posune jeden bit doleva. Algoritmus násobení produkuje výsledek na $n = n_1 + n_2$ bitech, kde n_1 a n_2 je velikost vstupů. Pro uchování velikosti aritmetiky je výsledek vždy oříznut. Oříznutí probíhá s ohledem na pozici řádové čárky, která se po násobení také posune na pozici $k = k_1 + k_2$, kde k_1 a k_2 jsou pozice řádových čárek ve vstupech.

$b_0 b_{-1}$	akce
00	nic
01	přičti násobitele
10	odečti násobitele
11	nic

Tabulka 8.1: Rozhodovací tabulka násobení [15]

Tabulka 8.2 zobrazuje ukázku násobení dvou čísel s desetinnou čárkou $(010.0)_2 = (2)_{10}$ a $(001.1)_2 = (1.5)_{10}$. Jako násobence použijeme číslo (001.1). Na konec násobence je potřeba přidat číslo nula. Výsledkem je číslo $(011.0)_2 = (3)_{10}$

iterace	akce		$...b_0$	b_{-1}
1	10 - odčítání	0000	0011	0
1		0100		
1		1100	0011	0
1	posun	1110	0001	1
2	11 - nic	1110	0001	1
2	posun	1111	0000	1
3	01 - přičítání	1111	0000	1
3		0100		1
3		0011	0000	1
3	posun	0001	1000	0
4	00 - nic	0001	1000	0
4	posun	0000	1100	0
-	připsání čárky	0000	11.00	0

Tabulka 8.2: Násobení

8.3.3 Dělení

Pro implementaci dělení jsem využil algoritmus SRT popsaný v publikacích [8], a [15]. Zkratka tohoto algoritmu je odvozena od iniciálů jmen autorů. Algoritmus byl prezentovaný okolo roku 1995. Algoritmus SRT byl prvním prezentovaným algoritmem, který dokázal počítat se zápornými čísly. U algoritmů objevených dříve bylo nutné algoritmus provádět nad čísly s absolutní hodnotou. Po skončení výpočtu bylo následně upraveno znaménko výsledku.

SRT dělení používá čísla v plovoucí řádové čárce, proto je nutné zajistit převod. Pro dělitel D musí platit $0.5 \leq |D| \leq 1$. Dále pro dělenec R musí platit $R < |D|$. Algoritmus se rozhoduje podle tabulky a třech nejvyšších bitů průběžného zbytku. Sofistikovanější verze algoritmu využívají k rozhodování více bitů průběžného zbytku. V každé iteraci se rozhoduje zda se dělitel přičte, odečte, nebo se nic neprovede. Na konci každé iterace se průběžný zbytek posune doleva. Potřebný počet iterací je určen podle velikosti datového

typu. Číslo v plovoucí řádové čárce je možné zapsat jako $m * 2^e$. Pokud mámě dvě čísla $a = m_1 * 2^{e_1}$ a $b = m_2 * 2^{e_2}$, pak operace dělení a/b lze vyjádřit jako $m_1/m_2 * 2^{e_1 - e_2}$. Ukázka výpočtu dělení dělence $0,01111 * 2^5 = 15$ a dělitele $0,11 * 2^2$ je v tabulce 8.4. Výsledkem je číslo $1 - 11 - 100 * 2^{5-2}$, které se překódováním přepíše jako $0.10100 * 2^3 = 5$. Výsledek lze za běhu překódovat pomocí algoritmu popsaného v publikaci [8].

$r_n r_{n-1} r_{n-2}$	D > 0 Bit podílu	D > 0 operace	D < 0 bit podílu	D < 0 operace
000	0	nic	0	nic
111	0	nic	0	nic
001	1	-d	-1	+d
010	1	-d	-1	+d
011	1	-d	-1	+d
101	-1	+d	1	-d
110	-1	+d	1	-d
111	-1	+d	1	-d

Tabulka 8.3: Rozhodovací tabulka dělení [15]

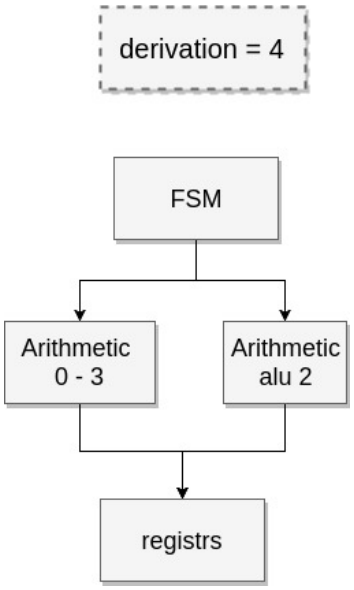
iterace	akce	$r_n r_{n-1} r_{n-2}$		doplnění	výsledek
0	vytvoření čísla 15	0.01	111	00000	
1	001 - odčítání	0.01	111	00000	1
1	-3 (1.01)	0.11	000		
1		1.10	111	00000	
1	posun	1.01	110	0000x	
2	101 - přičítání	1.01	110	0000x	-1
2	+3 (0.11)	0.11	000		
2		0.00	110	0000x	
2	posun	0.01	100	000xx	
3	001 - odčítání	0.01	100	000xx	1
3	-3 (1.01)	0.11	000		
3		1.10	100	000xx	
3	posun	1.01	000	00xxx	
4	101 - přičítání	1.01	000	00xxx	-1
4	+3 (0.11)	0.11	000		
4		0.00	000	00xxx	
4	posun	0.00	000	0xxxx	
5	000 - nic	0.00	000	0xxxx	0
5	posun	0.00	000	xxxxx	
5	000 - nic	0.00	000	xxxxx	0
6	zbytek = 0	0.00	000	xxxxx	

Tabulka 8.4: dělení

8.4 Integrátor

Komponenta slouží k aproximaci příští hodnoty integrálu z hodnot funkce pomocí Taylorova polynomu. Jednotka obsahuje několik jednotek *arithmetic* umožňujících sčítání, odčítání, násobení a dělení. Počet jednotek je závislý na počtu počítaných derivací. Jednotky pracují paralelně podle tabulky 4.1. Operace v tabulce 4.2 se provádí již sekvenčně. Výpočet koeficientu $k_i = \frac{j^i}{h^{i-1}}$ je prováděn sekvenčně po ukončení výpočtu zobrazeného v tabulce 4.2. Pokud by výpočet probíhal paralelně s výpočtem z tabulky 4.2. Na integrování by bylo potřeba asi o $\frac{1}{3}$ kratší dobu.

V komponentě je jednodušší protokol pro předávání vstupních a výstupních hodnot, jako v komponentách *LIM*, *FCE*. komponenta obsahuje dva signály. Signál *WORKIGN* pokud je v jedničce, tak komponenta integruje. Po nastavení signálu do nuly je možné přebrat si výsledek na výstupu *INTEGRAL*. Pro start integrování je nutné aby komponenta nepracovala a tedy měla signál *WORKIGN* nastaven do nuly. Po vložení dat na vstup derivací a počáteční hodnoty je nutné pro start integrace nastavit signál *START* do jedničky. V následujícím taktu nastavit signál *START* do nuly. V témže taktu dojde k nastavení signálu *WORKING* do logické jedničky. Obrázek 8.3 zobrazuje *integrátor* integrující pomocí pěti derivací. Čtyři jednotky arithmetic 0-3 aproximují jednotlivé derivace $f^{1-4}(x)$. Jednotka *arithmetic alu 2* slouží k vytvoření koeficientů $k_i = \frac{step\_int^{i+1}}{step\_dif^i}$.



Obrázek 8.3: integrator.vhd

8.5 Pamět

Nejdůležitější proměnné systému jsou zobrazeny na ukázce kódu níže. Proměnná x ukládá aktuální pozici hodnotu proměnných. Proměnná *x_save* slouží k ukládání hodnot x při výpočtu derivací. Na konci integrování je k hodnotě uložené v proměnné *x_save* přičten integrační krok. Výsledek je uložen zpátky do proměnné x. Proměnná *x_end* určuje konec integrace. Pokud $x_i >= x\_end_i$ pak dojde k ukončení výpočtu dané úrovně integrálu. Proměnné *step_int* resp. *step_dif* určují velikost integračního resp. derivačního kroku v

každé úrovni integrálu. Proměnná *y_val* ukládá mezivýsledky jednotlivých úrovní integrálu. Proměnné *dim* určuje aktuálně počítanou úroveň integrálu. Nakonec proměnná **der** určuje aktuálně počítanou derivaci pro jednotlivé úrovně integrálu.

```
signal step_int : arr_my_type(0 to dimension-1);
signal step_dif : arr_my_type(0 to dimension-1);
signal x        : arr_my_type(0 to dimension-1);
signal x_save   : arr_my_type(0 to dimension-1);
signal x_end    : arr_my_type(0 to dimension-1);

//vysledky jednotlivych podintegralu
type t_val is array (0 to dimension) of arr_my_type(0 to derivation-1);
signal y_val    : t_val;

//aktualne pocitany podintegral
signal dim : unsigned(log2(dimension+1)-1  downto 0);
//derivace v jednotlivych integralech
type t_der is array(0 to dimension) of unsigned(log2(derivation)-1 downto
signal der : t_der;
```

8.6 Vytvoření systému počítající specifický integrál

Pro vytvoření systému řešícího speciální integrál je nutné nastavit konstantu *dimension* určující rozměr integrálu. Dále je nutné změnit implementaci komponentám *LIM* a *FCE*. Komponenta *LIM* zajišťuje výpočet horní a dolní meze jednotlivých úrovní integrálů. Komponenta *FCE* se stará a výpočet zadané funkce v integrálu. Komponenty jenom musí dodržovat rozhraní. Předávání vstupu a výstupu probíhá stejně jako u datového typu. Na začátku i na konci výpočtu se parametry přebírají pomocí dvojice signálu *DST_RDY* a *SRC_RDY*. Podle toho, zda se jedná o komunikaci na začátku resp. na konci, mají signály předponu *IN* resp. *OUT*. Signál *X* přestavuje aktuální hodnotu proměnné $\vec{x}$. Někdy také reprezentované, jako $x_0 = x$, $x_1 = y$, $x_2 = z$, a další. Tyto signály jsou společné pro obě komponenty.

Komponenta *LIM* obsahuje navíc signály *DIM*, *X_START*, a *X_END*. Signál *DIM* slouží k určení pro kterou úroveň integrálu se bude počítat horní a dolní mez. Výstupní signál *X_START* je horní mezí aktuální úrovně integrálu. Nakonec výstupní signál *X_END* je dolní mezí aktuální úrovně integrálu.

Komponenta *FCE* obsahuje navíc výstupní signál *OUT_DATA*. Hodnota tohoto signálu je na konci výpočtu nastavena na hodnotu integrované funkce pro proměnou $\vec{x}$ předanou signálem *X*.

```
entity lim is
generic(
        DIMENSION : integer
);
port (
        CLK                 : in  std_logic;
        RESET               : in  std_logic;
        -- ------------------------------------
        IN_SRC_RDY          : in  std_logic;
        IN_DST_RDY          : out std_logic;

        X                   : in arr_my_type(0 to DIMENSION-1);
        DIM                 : in unsigned(log2(DIMENSION) downto 0);

        OUT_SRC_RDY         : out std_logic;
        OUT_DST_RDY         : in  std_logic;

        X_START             : out my_type;
        X_END               : out my_type
);
end entity lim;

entity fce is
generic (
        DIMENSION : integer
);
port (
        CLK                 : in  std_logic;
        RESET               : in  std_logic;
        -- ------------------------------------
        IN_SRC_RDY          : in  std_logic;
        IN_DST_RDY          : out std_logic;
        X                   : in  arr_my_type(0 to DIMENSION-1);

        OUT_SRC_RDY         : out std_logic;
        OUT_DST_RDY         : in  std_logic;
        OUT_DATA            : out my_type);
end entity fce;
```

Pro příklad jsem ve zdrojovém kódu *system.vhd* zvolil implementovat integrál (8.1). Kód komponent *LIM* a *FCE* je zobrazen v příloze. V tomto případě je hodnota konstanty *dimension* nastavena na hodnotu 2.

$$\int_1^{10}\int_1^{10} x^2 dxdy \tag{8.1}$$

8.7 Simulace a překlad

Simulace se spouští v adresáři *vhdl/sim* příkazem *vsim -do vhdl.fdo*. Simulace se spustí s předefinovaným problémem. Nepřesnost výsledku v simulaci je způsobena malou přesností zadané inverzní matice.

$$\int_1^{10}\int_1^{10} x^2 dxdy$$

Překlad se spouští v adresáři *vhdl/synth* příkazem *make*. Důležitým výstupem jsou dva soubory *synth.tim* a *synth.util*. V prvním souboru se nachází souhrn o časování. V druhém výstupu se nachází souhrn o zabraných zdrojích na použitém čipu. Nevýhodou mé implementace je velká spotřeba zdrojů modulu *DIV*. Modul spotřebovává nejvíce zdrojů při převodu z datového typu s pevnou řádovou čárkou na datový typ s pohyblivou řádovou čárkou. Tento problém jsem částečně vyřešil převodem ve více hodinových taktech, ale modul má stále velký vliv na počet zabraných LUT jednotek. Problém je více patrný při větších datových typech. Datový typ v hardwaru uvádím jako x/y, kde x představuje celkový počet bitů a y počet bitů za desetinnou čárkou. Jak je vidět při zvětšení datového typu enormně narůstá počet zdrojů, tento jev je nutné pro budoucí použití odstranit. Snížení náročnosti na počet LUT je zavedením postupného sčítání. V tabulce je zobrazen případ, kdy se již obvod nevleze do malého čipu a potřebuje 4-krát více LUT jednotek. Hodnota LUT 2 v tabulce 8.5 vznikla odstraněním některých děliček z návrhu. Odstranění děliček bylo možně převodem operace dělení na operaci násobení. Došlo k nahrazení konstanty *to_my_type(I+2)* konstantou *to_my_type(1.0/(I+2))*.

Čip	Datavý typ	velikost matice	LUT (%)	registry(%)	LUT 2 (%)
xc7k70tfbg676-2	64/48	4	24240 (59%)	7932(9%)	11312(27%)
xc7k70tfbg676-2	128/64	4	54221 (132%)	14622(18%)	19015(46%)
xc7k70tfbg676-2	256/128	4	121668(296%)	28872(35%)	52773(128%)
xc7k70tfbg676-2	64/48	5	28318 (69%)	8624 (11%)	12169(29%)
xc7k70tfbg676-2	128/64	5	64100 (156%)	11312(15%)	20832(50%)
xc7k70tfbg676-2	256/128	5	142428(347%)	31758(38%)	56410(137%)
xc7k70tfbg676-2	64/48	7		8505(10%)	13831(33%)
xc7k70tfbg676-2	128//64	7		16715(20%)	27005(65%)
xc7k70tfbg676-2	256/128	7		32942(40%)	62898(153%)

Tabulka 8.5: Tabulka spotřeby zdrojů HW implementace

8.8 Zrychlení

V posledním experimentu měřím zrychlení oproti softwarové implementaci. U softwarové implementace je potřebný čas změřen pomocí funkce *clock*. U hardwarové implementace je potřebný čas změřen pomocí kombinace překladu a simulace. Při překladu jsem zjistil možné nastavení hodinového signálu. Následně jsem v simulaci nastaven hodinový signál tak aby odpovídal hodinovému signálu z překladu. Výsledný čas je určen simulačním časem, kdy je ukončena práce obvodu. Pro účely testu byl zvolen integrál.

$$\int_1^{10}\int_1^{10} x^2 dxdy = 2997$$

Pro tento test jsem zvolil integrační krok 1.0 a derivační krok 0.125. V tabulce je zobrazena rychlost v softwaru a hardwaru.

SW	sw typ	HW	hw typ
0.000695s	double	0.003s	64/48
0.004530s	gmp-256	0.014s	256/128

Tabulka 8.6: Tabulka času SW a HW implementace

Kapitola 9

Závěr

V této diplomové práci je popsán princip využití Taylorovy řady pro výpočet integrálů vyšších řádů s prostorovou proměnnou. Byly navrženy dva systémy pro výpočet integrálů: softwarový a hardwarový. Hardwarová varianta je daleko méně přesná díky převodu inverzní matice z vestavěného datového typu *real* ve VHDL. To je zejména patrné na výsledcích. Pro přesnou matici by výsledky odpovídaly přesnosti softwarové implementaci. Práce uvádí dále také několik možností paralelizace Taylorovy řady pro výpočet vícenásobných integrálů. Při návrhu obou architektur jsem dbal na modularitu a jednoduchou rozšířitelnost.

Bylo provedeno velké množství experimentů, ve kterých bylo dokázáno, že Taylorova metoda je velice přesná. Pro integrování funkcí s počtem derivací menším než je počet počítaných derivací je metoda absolutně přesná. Výpočet je efektivnější s vysokým počtem derivací a velkým krokem. Oba systémy umožňují výpočet různých vícenásobných integrálů pro prostorové proměnné.

Budoucím vylepšením může být přechod od datového typu s pevnou řádovou čárkou k datovému typu s plovoucí řádovou čárkou. U obou variant lze využít Boothovo překódování. Zajímavou změnou by bylo, kdyby se výpočet funkce prováděl paralelně pro všechny uzlové body. Paralelně by se měly počítat jen funkce, které jsou potřebné pro krok integrátoru. Dále by se implementace integrátoru, funkce a výpočet limit přesunul do mikrokódu. Nad stejnou jednotkou by se vykonávalo integrování, výpočet funkcí a výpočet limit jednotlivých úrovní integrálu. Vylepšení by se ušetřil využitý prostor na čipu.

Literatura

[1] A. Abad, F. B. M. R., R. Barrio: TIDES, a Taylor Series Integrator for Differential EquationS.

[2] B.P.Děmidovič; I.A.Maron: *Základy numerické matematiky*. SNTL, 1966.

[3] Dlouhý, Z.; Hruška, K.; Kůst, J.; aj.: *Úvod do matematické analýzy*. Státní pedagogické nakladatelství, 1965, publikace č. 45-00-05.

[4] Drábek, V.: *Výstavba počítačů*. Vysoké učení technické v Brně, 1995, iSBN-80-214-0691-7.

[5] Jarník, V.: *Integrální Počet (I)*. Academia, 1984.

[6] Kalas, J.; Kuben, J.: *Integrální počet funkcí více proměnných*. Masarikova univerzita, 2009, iSBN 978-80-210-4975-8.

[7] Kubíček, M.; Dubcová, M.: *Numerické metody a algoritmy*. VŠCHT praha, 2005, iSBN-80-7080-558-7.

[8] MATEČNÝ, F.: *SIMULÁTOR PROCESORA S OPERÁCIOU DELENIA*. Vut Brno, 2016.

[9] Mikulka, J.: *Numerické výpočty určitých integrálů*. Vut Brno, 2014.

[10] Míka, S.: *Numerické metody a algebry*. SNTL, 1985.

[11] Nečasová, G.: *Parciální numerické řešení parciálních diferenciálních rovnic*. Vut Brno, 2014.

[12] Nýč, P.; EISLER, J.: *Tabulky neurčitých integrálů*. en, 1991, iSBN-80-901070-0-1.

[13] Opálka, J.: *Automatické řízení výpočtu ve specializovaném výpočetním systému*. Vut Brno, 2016.

[14] Peringer, P.: Simulační nástroje a techniky [online]. `https://www.fit.vutbr.cz/study/courses/SNT/public/Prednasky/SNT.pdf`, 2016-02-08 [cit. 2016-05-18].

[15] Sekanina, L.: slajdy k predmetu INP - Návrh počítačových systémů. 2015.

[16] Shi, K.; Boland, D.; Constatinides, G. A.: Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators. In *Field-Programmable Technology (FPT), 2014 International Conference on*, editace editor.

[17] WWW stránky: The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`.

[18] Šátek, V.: *Analýza stiff soustav diferenciálních rovnic*. Vut Brno, 2011.

Příloha A

Ukázka definování problému

$$\int_{10}^{26} \int_{0}^{y} \frac{x^2}{y} dxdy$$

```
//////////////////////////////////////////////////////
//PROBLEM int int (x^2/y) from 0 to y dx from 1 to 100 dy
//////////////////////////////////////////////////////
//prozatim jenom sinus potom opravit na neco lepsiho
#define PROBLEM_4_DIM 2
//                                              0-x     1-y
static long double problem_4_step_diff[] = {0.5L, 0.5L};
static long double problem_4_step_int[]  = {2.0,  2.0};

#define PROBLEM_4_CONST_SIZE 4
static long double problem_4_const[] = {0.0L, 10.0L, 26.0L, 3.0L};

//array of instruction for function
static int_equation_prog problem_4_fce[]= {
//start
{INT_EQ_INSTRUCTION_START , 2, 0, 0}, //get elements on stack
{INT_EQ_INSTRUCTION_X_TO_LOC, 0, 0, 0}, //load actual x
{INT_EQ_INSTRUCTION_MUL    , 0, 0, 0},   // tmp[0] = x^2
{INT_EQ_INSTRUCTION_X_TO_LOC, 1, 1, 0}, // tmp[1] = y
//end retun value save in data[0]
{INT_EQ_INSTRUCTION_DIV    , 0, 0, 1},  // x^2/y tmp[0] = tmp[0]/tmp[1]
{INT_EQ_INSTRUCTION_RETURN, 0, 0, 0}
};


//array of function definition start and end of integral x
static int_equation_prog  int_4_step_0[]= {
//start
{INT_EQ_INSTRUCTION_START , 1, 0, 0},
//setup start x
{INT_EQ_INSTRUCTION_CONST_TO_LOC, 0, 0, 0},  //load 0.0L from const[0]
{INT_EQ_INSTRUCTION_LOC_TO_X, 0, 0, 0},      // to x_0
```

```
//setup end x
{INT_EQ_INSTRUCTION_X_TO_LOC, 0, 1, 0},     //load Y
{INT_EQ_INSTRUCTION_LOC_TO_XEND, 0, 0, 0}, // to x_end_0
//end
{INT_EQ_INSTRUCTION_RETURN, 0, 0, 0}
};

//array of function definition start and end of integral y
static int_equation_prog  int_4_step_1[]= {
//start
{INT_EQ_INSTRUCTION_START , 1, 0, 0},
//setup start x
{INT_EQ_INSTRUCTION_CONST_TO_LOC, 0, 1, 0},  // load 1.0L from const[1]
{INT_EQ_INSTRUCTION_LOC_TO_X, 1, 0, 0},      // to x_1
//setup end x
{INT_EQ_INSTRUCTION_CONST_TO_LOC, 0, 2, 0}, // load  from const [2]
{INT_EQ_INSTRUCTION_LOC_TO_XEND, 1, 0, 0},  // to x_end_1
//end
{INT_EQ_INSTRUCTION_RETURN, 0, 0, 0}
};


//problem int(int(sin(x))dx)dy;
bool problems_4_init(t_problem * init){

//setup num of dimensions
init->dim = PROBLEM_4_DIM;
//create function
init->fce = problem_4_fce;
// create equention for creating inegral border for every sub integral
init->int_start    = malloc(sizeof(void*)* PROBLEM_4_DIM);
init->int_start[0] = int_4_step_0;
init->int_start[1] = int_4_step_1;

//create space for x, step_dif and step int
init->x          = malloc(sizeof(int_variable) * PROBLEM_4_DIM);
init->x_end      = malloc(sizeof(int_variable) * PROBLEM_4_DIM);
init->step_diff  = malloc(sizeof(int_variable) * PROBLEM_4_DIM);
init->step_int   = malloc(sizeof(int_variable) * PROBLEM_4_DIM);
//init memory
int_variable_arr_init(init->x         , PROBLEM_4_DIM);
int_variable_arr_init(init->x_end     , PROBLEM_4_DIM);
int_variable_arr_init(init->step_diff, PROBLEM_4_DIM);
int_variable_arr_init(init->step_int , PROBLEM_4_DIM);
//setup variable
for(int i = 0; i < PROBLEM_4_DIM; i++){
int_variable_create(init->step_diff + i, problem_4_step_diff[i]);
int_variable_create(init->step_int  + i, problem_4_step_int[i]);
```

```
}

//if coumputing need some constants
init->constants = malloc(sizeof(int_variable) * PROBLEM_4_CONST_SIZE);
int_variable_arr_init(init->constants, PROBLEM_4_CONST_SIZE);
for(int i = 0; i < PROBLEM_4_CONST_SIZE; i++){
int_variable_create(init->constants + i, problem_4_const[i]);
}
return true;
}
```

Příloha B

HW implementace LIM a FCE

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.math_pack.all;
use work.type_pack.all;
use work.integrator_pack.all;


entity lim is
generic(
DIMENSION : integer
);
port (
CLK                 : in  std_logic;
RESET               : in  std_logic;

-- ------------------------------------------------------------------------
IN_SRC_RDY          : in  std_logic;
IN_DST_RDY          : out std_logic;

X                   : in arr_my_type(0 to DIMENSION-1);
DIM                 : in unsigned(log2(DIMENSION) downto 0);


OUT_SRC_RDY         : out std_logic;
OUT_DST_RDY         : in  std_logic;

X_START             : out my_type;
X_END               : out my_type
);
end entity lim;

architecture FULL of lim is
begin
```

```
--no limit for DIM
IN_DST_RDY  <= OUT_DST_RDY;
OUT_SRC_RDY <= IN_SRC_RDY;

X_START <= to_my_type(1.0);
X_END   <= to_my_type(10.0);

end architecture FULL;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.type_pack.all;
use work.math_pack.all;
use work.integrator_pack.all;

entity pow2 is
generic (
DIMENSION : integer
);
port (
CLK                 : in  std_logic;
RESET               : in  std_logic;
-- ------------------------------------------------------------------------
IN_SRC_RDY          : in  std_logic;
IN_DST_RDY          : out std_logic;
X                   : in  arr_my_type(0 to DIMENSION-1);

OUT_SRC_RDY         : out std_logic;
OUT_DST_RDY         : in  std_logic;
OUT_DATA            : out my_type);
end entity pow2;

architecture FULL of pow2 is
type t_fsm is (S_STOP, S_MUL_START, S_MUL, S_WAIT);
signal fsm : t_fsm := S_STOP;
signal fsm_next : t_fsm;


signal alu_op           : std_logic_vector(1 downto 0);
signal alu_in_src_rdy     : std_logic;
signal alu_in_dst_rdy     : std_logic;
signal alu_in_data_1    : my_type;
signal alu_in_data_2    : my_type;
signal alu_out_src_rdy  : std_logic;
signal alu_out_dst_rdy  : std_logic;
signal alu_out_data     : my_type;

signal sig_set_out      : std_logic;
signal sig_set_in       : std_logic;
signal x_in             : my_type;

signal reg_out          : my_type;
begin
```

```
PROC_FSM : process(CLK)
begin
if(CLK = '1' and CLK'event) then
if(RESET = '1') then
fsm <= S_STOP;
else
fsm <= fsm_next;
end if;
end if;
end process;


PROC_FSM_NEXT : process(all)
begin
fsm_next <= fsm;
case fsm is
when S_STOP =>
if (IN_SRC_RDY = '1') then
fsm_next <= S_MUL_START;
end if;

when S_MUL_START =>
if(alu_in_dst_rdy = '1') then
fsm_next <= S_MUL;
end if;

when S_MUL =>
if(alu_out_src_rdy = '1') then
fsm_next <= S_WAIT;
end if;

when S_WAIT =>
if(OUT_DST_RDY = '1') then
fsm_next <= S_STOP;
end if;
end case;
end process;


PROC_FSM_OUT : process(all)
begin
IN_DST_RDY  <= '0';
OUT_SRC_RDY <= '0';
sig_set_out <= '0';
sig_set_in  <= '0';
alu_in_src_rdy <= '0';

case fsm is
```

```
when S_STOP =>
IN_DST_RDY <= '1';
sig_set_in <= '1';

when S_MUL_START =>
alu_in_src_rdy <= '1';
alu_op         <= "10";
alu_in_data_1  <= x_in;
alu_in_data_2  <= x_in;

when S_MUL =>
sig_set_out <= '1';


when S_WAIT =>
OUT_SRC_RDY <= '1';

end case;
end process;


REG : process(CLK)
begin
if(CLK = '1' and CLK'event) then
if(sig_set_out = '1') then
reg_out <= alu_out_data;
end if;

if(IN_SRC_RDY <= '1' and  sig_set_in = '1') then
x_in <= X(0);
end if;
end if;
end process;


ALU : entity work.aritmetic
port map(
CLK                 => CLK,
RESET               => RESET,

OP                  => alu_op,

IN_SRC_RDY          => alu_in_src_rdy,
IN_DST_RDY          => alu_in_dst_rdy,
IN_DATA_1           => alu_in_data_1,
IN_DATA_2           => alu_in_data_2,
```

```
OUT_SRC_RDY         => alu_out_src_rdy,
OUT_DST_RDY         => alu_out_dst_rdy,
OUT_DATA            => alu_out_data
);
alu_out_dst_rdy <= '1';

OUT_DATA <= reg_out;
end architecture FULL;
```

Příloha C

Obsah CD

- latex - Zdrojové soubory této práce ve formátu LaTeX
- src - zdrojové soubory integrátoru v SW verzi
- vhdl - zdrojové soubory integrátoru v HW verzi
- text této práce ve formátu PDF